Dissertation

Christian Hörtagl
Evaluation of Technologies
of Parallel Computers’ Communication Networks
for a Real-Time Triggering Application
in a High-Energy Physics Experiment at CERN

Dissertation
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften

ausgeführt am
European Laboratory for Particle Physics, CERN
und

ingereicht an der
Naturwissenschaftlichen Fakultät
der
Leopold-Franzens-Universität Innsbruck

von
Dipl.-Ing. Christian Hörtnagl

Genf und Innsbruck, im Dezember 1997
Abstract

I wonder why I wonder why. I wonder why I wonder.
I wonder why I wonder why I wonder why I wonder.

RICHARD P. FEYNMAN

Experiments at the future Large Hadron Collider (LHC) at CERN will be faced with an extraordinary challenge of event selection in real time. The primary event rate, equal to the bunch crossing frequency of 40 MHz, will have to be reduced by a factor of almost one-in-a-million in order to reveal traces of rare physics processes from an abundant background.

This work presents various contributions to ongoing feasibility studies concerning the possible use of commercial technologies from the proximities of parallel computers and their communication networks for the second trigger stage, which faces an average data input rate of 100 kHz.

Studies in this thesis apply a combination of methodologies, namely the build-up of lab-scale prototype implementations (including their exposition to test beam runs), algorithm development, technology tracking and benchmarking, as well as discrete event simulation.

The main contribution consists of several technology case studies, which are based on the exploration of a set of standard benchmark programs for revealing simple parameters for characterizing delays during communication. Studied technologies include the communication sub-system of the Meiko CS-2, Asynchronous Transfer Mode (ATM), MEMORY CHANNEL, and Scalable Coherent Interface (SCI); all could be considered typical for candidate technologies.

The discussion sheds light on the relative benefits and costs associated with different parallel programming models, in general, and with the use of message-passing libraries, such as Message Passing Interface (MPI), in particular.

Best observed end-user-to-end-user latencies were \( \sim 10 \ \mu s \), best asymptotic bandwidths were \( \sim 70 \ \text{MByte/s} \). Typical sub-patterns of communication that have to be applied in the second trigger stage were sustained at \( \gtrsim 13 \ \text{kHz} \), using today’s technologies in realistic embeddings.
I am most indebted to my principal supervisor, Dr. Rudolf Böck (CERN), who invited me to join his team at CERN and therefore enabled the present doctoral thesis. He mastered whatever challenges arose with good humor and unshaking loyalty and helped with strong dynamism and support during the of course urgent finishing phase.

My academic supervisor, Prof. Dietmar Kuhn (University of Innsbruck), was a most sincere, prompt, and reliable source of optimism and administrative advice during all times of need. I attribute to him many of the good features that an academic group leader should ideally possess and appreciate his lasting help. I also thank the other members of the High Energy Physics group at the Institute for Experimental Physics in Innsbruck for the honor of allowing me to work in their ranks.

I very heartily thank Dr. Reiner Hauser (CERN and Digital Equipment Corporation), who was my most accessible senior professional colleague and indeed a good friend. I owe him much for general encouragement and specific explanations. For what is even more important to my mind, he helped to refine my strong reading habit by suggesting lots of now esteemed titles.

Many colleagues at CERN very properly exhibited a noble aspect of the organization’s international spirit that allows researchers from many institutions world-wide and from junior to very senior to collaborate on common goals. (CERN’s staff includes even Nobel Prize winners, as every new visitor is proudly informed, when one of them makes his appearance in the cafeteria.) At a time when the World Wide Web began its boom, and when new scientific collaborations were in the making, taking part was usually a great experience.

I thank Dr. Günther Dissertori (CERN), with whom I shared a flat in Geneva, for the good company over the past few years and for being a good friend. The cheerful crowd of his mainly Italian acquaintances accepted me in their lot and gave me the great pleasure of knowing almost twice as many friends in town than at other times.
I sincerely thank my family and friends from home, especially all those who visited me in Geneva. Special thanks go to my brother Dr. Andreas Hört nagl (then University of Innsbruck), who was instrumental in getting me back on the academic track after some years of professional work, and to my parents Ingrid and Andreas Hört nagl, who kindly extended childhood privileges to a grown-up son, while I was finishing my Ph.D.

I owe gratitude to those friends and fellow graduate students who joined me to places even as remote as the squat bars in Geneva and the highest alpine peaks. Our “scrambles among the alps”, in particular, will remain as long-lasting memories from my stays in France and Switzerland.

I am also indebted to Dr. Luitwin Sch arfetter (then CERN and University of Innsbruck), for his hospitality, and to Prof. Chris Fabjan (CERN and Vienna University of Technology) and Prof. Peter Girtler (University of Innsbruck), for their willingness to serve on the board of referees.

Last but not least I respectfully acknowledge help from the authorities at CERN and at the Austrian Federal Ministry for Science, Transports, and the Arts, for allowing me to pursue my studies under a research grant from their CERN Doctoral Students program.

Editorial note Although the main contributions to the thesis stem from practical work during the past few years at CERN, I wanted to accomplish an extra goal when writing up this final report. Today it is still far from usual (if possible) that Austrian students deliver their written theses in English (instead of German), although I personally think that the use of English is highly desirable and appropriate for work that is done on the Ph.D. level. When I set out to implement the goal, I held some hope that readers with a firmer command of the language than myself would be willing to allow for some extra tolerance in cases where slight grammatical inconsistencies may have remained.

Christian Hört nagl
December, 1997
Contents

Abstract i

Acknowledgments iii

Contents vii

List of figures xi

1 The ATLAS experiment 1
  1.1 The Large Hadron Collider (LHC) 1
  1.2 The ATLAS detector 3
    1.2.1 Semi-Conductor Tracker (SCT) 6
    1.2.2 Transition Radiation Tracker (TRT) 6
    1.2.3 Calorimeter 7
    1.2.4 Muon spectrometer 7
  1.3 Trigger and Data Acquisition System (DAQ) 8
    1.3.1 LVL1 trigger 10
    1.3.2 LVL2 trigger 13
    1.3.3 LVL3 trigger (event filter) 14
    1.3.4 Implementation choices for the 100 kHz trigger 15
    1.3.5 Building blocks of the LVL2 trigger 17
      1.3.5.1 Supervisor 17
      1.3.5.2 Readout buffers 18
      1.3.5.3 Feature extraction processors 20
      1.3.5.4 Global processing 20
  1.3.6 Architecture A 21
  1.3.7 Architecture B 22
  1.3.8 Architecture C 24
  1.3.9 Quantitative comparison between readout formats for the end-cap TRT 25
2 Aspects of a specific 1996 technological implementation
2.1 An implementation of a TRT feature extraction algorithm for DSP320C4x Digital Signal Processors
2.2 A VMEbus-based server for DSP320C4x Digital Signal Processors
   2.2.1 DSP320C4x Digital Signal Processors
   2.2.2 VMEbus boards with DSPs
   2.2.3 SBus-to-VMEbus interface
   2.2.4 COFF files
   2.2.5 Protocol for down-loading executables
   2.2.6 Protocol for receiving output from client programs
   2.2.7 Command line options

3 Technology tracking and communication benchmarks
3.1 A taxonomy of multiprocessor architectures
   3.1.1 Shared memory programming model
      3.1.1.1 Uniform memory access
      3.1.1.2 Non-uniform memory access
   3.1.2 Message passing programming model
      3.1.2.1 MPI – Message Passing Interface
3.2 ATLAS communication benchmarks
   3.2.1 Low level interface
   3.2.2 Towards a possible one-copy implementation
3.3 Technology case studies
   3.3.1 The communication sub-system of the Meiko CS-2
      3.3.1.1 An implementation of MPI for Elan channels and the MPICH channel interface
   3.3.2 ATM – Asynchronous Transfer Mode
   3.3.3 Memory Channel
      3.3.3.1 Maximum sustained bandwidth
      3.3.3.2 Results for ATLAS communication benchmarks with and without Digital MPI
   3.3.4 SCI – Scalable Coherent Interface
      3.3.4.1 Software configuration
      3.3.4.2 Implementation of the low level interface
      3.3.4.3 Results for ATLAS communication benchmarks
      3.3.4.4 Conclusions
   3.4 Lessons learned from benchmarking

4 Modeling as a tool for optimization
4.1 Scope of modeling and simulation
   4.1.1 Role in top-down analysis
   4.1.2 Role in bottom-up designs
4.2 Discrete event simulation ........................................ 151
  4.2.1 Clock-driven approach .................................. 153
  4.2.2 Event-driven approach .................................. 153
  4.2.3 Process-oriented discrete event simulation .......... 155
4.3 SIMDAQ — A generic model of the ATLAS LVL2 trigger .... 156
  4.3.1 MODSIM-II ............................................. 158
  4.3.2 A model of the C104 Asynchronous Packet Switch .... 159

Glossary of acronyms  ............................................. 171

Bibliography ......................................................... 175
List of Figures

1.1 the ATLAS detector ........................................ 3
1.2 some trigger menu items at high luminosity ................. 12
1.3 detector readout parameters ................................ 12
1.4 effects of partitioning tasks on overall required effort .... 16
1.5 $\eta \times \phi$ cut-away view of the “unrolled” end-cap TRT .... 26
1.6 candidate readout formats for the end-cap TRT ............... 28
1.7 output formats before feature extraction in the end-cap TRT ... 28
1.8 pre-processing in the end-cap TRT .......................... 30

2.1 proposed setup for lab tests at CERN in 1996 ............... 33
2.2 a sample event from the end-cap TRT prototype ............... 35
2.3 high-level protocol employed over HIPPI lines ............... 37
2.4 readout sequence for straws in left and “right” roofs ....... 37
2.5 features extracted from the end-cap TRT prototype .......... 39
2.6 protocol for down-loading software via communication ports . 42
2.7 RAM options for on-chip and on-module memory .............. 43
2.8 RAM options for on-board memory .......................... 44
2.9 field values for Global Memory Interface Control Register ... 46
2.10 clients and servers in the test lab setup .................... 46
2.11 line syntax for topology description files .................. 50
2.12 protocol for communication from server to clients ........... 51
2.13 prototypes defined in $stdio.h$ ............................ 52
2.14 summary of command line options supported by $sunvme$ .... 54

3.1 growth in microprocessor performance over the past decade ... 59
3.2 evolutionary versus revolutionary trends in computer design .. 60
3.3 a taxonomy of multiprocessor memory architectures .......... 62
3.4 memory hierarchy in the AlphaServer 8400 .................. 64
3.5 minor communication modes in MPI ........................ 74
3.6 global communication operations in MPI .................... 76
3.7 global reduction operations in MPI ........................ 76
3.8 ATLAS communication benchmarks .......................... 81
3.9 studies of network interference with $mpirun(1)$ .......... 83
3.10 protocol stack used in the example implementation .......... 85
one layer of a three-stage CS-2 communication network
MPICH protocol stack and the OSI reference model
protocol suites with and without MPICH-specific layers
results for Meiko CS-2 with and without MPICH-specific protocol layers
ATM performance measurements
embedding of MEMORY CHANNEL address space in other address spaces
maximum sustained bandwidths
overview of cluster-wide shared data structures
schematic view of synchronous and asynchronous queues
class configurations for MEMORY CHANNEL measurements
results for configuration 1
results for configuration 1 with Digital MPI
results for configuration 2
results for configuration 2 with Digital MPI
results for configuration 3
results for configuration 3 with Digital MPI
results for configuration 4
results for configuration 4 with Digital MPI
early results for DAQ61 benchmark suite
basic sequence of SCI transactions
cache invalidation after SCI transactions
cluster configurations for SCI measurements
related standards governing shared memory access
protocol stacks for ATLAS communication benchmarks
overview of cluster-wide shared data structures
an example involving use of the SCI PHY-API
results from AlphaStation 500/400 to AlphaServer 4000 5/300
results from RIO 8061 to AlphaServer 4000 5/300
results from RIO 8061 to RIO 8061
results for message passing between different types of nodes
results for message passing in different topologies
results for DAQ61 benchmark suite
results for distributed vs. centralized shared memory systems
relationship between a real system and its model
process-oriented vs. event-oriented discrete event simulation
schematics of a detailed C104 model
scaling properties of different network topologies
a four-dimensional hypercube
a section of the GPMIMD machine
graphical notation for Fig. 4.8

x
4.8 an example of the C104 model’s functioning . . . . . . . . . . . . 168
4.9 comparison between the model’s and earlier results . . . . . . . . . . 169
Chapter 1

The ATLAS experiment

*The real goal of physics is to come up with an equation that can explain the universe, but is still small enough to fit on a T-shirt.*

Leon Lederman

*There are protons, electrons, and so-ons ...*

David Hawkings

1.1 The Large Hadron Collider (LHC)

The Large Hadron Collider (LHC) is a new particle accelerator, which is scheduled for operation at CERN’s site close to Geneva, Switzerland, starting from the year 2005. Unlike CERN’s largest existing accelerator ring, the Large Electron Positron Collider (LEP), LHC will boost beams of protons into head-on collisions at interaction energies of 7-on-7 TeV. (LEP2 ran at up to 90-on-90 GeV in 1996.) Since protons can be accelerated in circular machines to much higher energies than electrons, center of mass energies can be substantially increased at the collision point. On the other hand the resulting physics events impose considerably larger problems for reconstruction and analysis steps because of the internal structure of protons.

In this mode the new collider ring will provide a nominal top luminosity of at least $L = 10^{34}$ cm$^{-2}$ s$^{-1}$, for 2 proton-proton experiments running simultaneously. An alternative mode of operation will allow injection of heavy ions. LHC’s 2 anti-parallel accelerator rings will be fitted into the tunnel such that the LEP machinery can potentially be reinstalled on top of the LHC magnets at a later time, thus enabling an option for future electron-proton collision experiments.

The size and complexity of the ensuing technological challenges and the required financial resources call for international scientific collaboration of the kind for which CERN’s 19 member states have also provided the framework during past experiments in the institute’s over 40 years long history. Among the most successful was UA1 with its discovery of the $W^\pm$ and $Z^0$ bosons; it confirmed the
electroweak theory and won Carlo Rubbia and Simon Van der Meer the Nobel Prize in Physics in 1984.

CERN currently has an important competitive advantage over other candidate research sites, because a significant part of the required local infrastructure is potentially available for reuse from previous experiments. Among already payed-for utilities are a number of pre-accelerators and storage rings, whose increasing sizes (and thus energies) reflect CERN’s past research record, which allowed scientists to keep up with the predictions of contemporary theories and penetrate increasingly further into the structure of matter as more powerful experimental facilities were constructed over the years. Therefore particles that collide in LEP or LHC pass through a chain of accelerators (Linac at 50 MeV, Booster at 800 MeV, PS at 28 GeV, SPS at 450 GeV) that also represent past milestones in the history of HEP research.

When LHC will become operational, counter-rotating bunches of $\sim 10^{11}$ particles each will be injected into the rings with periods of 25 ns in between. With filling, they will undergo acceleration up to their nominal collision energies in about 20 minutes. The envisaged energies open new discovery domains at the highest energy frontiers; they will allow scientists to study phenomena on formerly unseen small scales, and reproduce the conditions that prevailed in the universe just briefly after the Big Bang.

CERN’s existing circular underground tunnel of 27 km circumference today houses the LEP accelerator. It crosses the border between neighboring Switzerland and France twice at depths of around 100 m below surface level. Because of the synchrotron radiation that is emitted by accelerated electrons and positrons, a ring of very large radius was required for LEP, in order to keep the total energy loss and consumption within manageable bounds. This tunnel is the most prominent asset from whose reuse LHC can profit in terms of cost-effective implementation.

As a consequence from this development strategy, extra challenges result for the design of the new magnet system, which holds the rotating beams on steady courses during acceleration, because the target energy must be achieved within the constraints of the existing tunnel. Unlike at LEP the magnet system will therefore operate in super-fluid helium at 2 K.

The current accelerator feeds the 4 LEP experiments ALEPH, DELPHI, L3, and OPAL, whose detectors are located in underground experimental halls at equidistant points around the circumference of the tunnel. Likewise, LHC will support 4 major experiments. In addition to the 2 high-luminosity proton-proton experiments ATLAS [5] and CMS [17], the ALICE and LHC-b collaborations have come up with proposals for experiments that are more specifically targeted at studying heavy ions (Pb — Pb) and a B-physics program respectively.
1.2 The ATLAS detector

The ATLAS collaboration [4, 5] proposes to build one of the 2 general-purpose proton-proton collision experiments that are foreseen for operation at the future LHC accelerator. (The “other” experiment is CMS [17]; shared design challenges result in a similar overall design.)

The shape of the ATLAS detector from outside view approximately matches a cylinder whose axis coincides with the horizontal beam-line in the underground cavity that houses the experiment. This is commonly the case with HEP collider experiments, as in particular also with the 4 presently existing LEP experiments at CERN (ALEPH, DELPHI, L3, and OPAL). However, due to its more ambitious design goals the ATLAS detector is quite exceptional in size, with a diameter of about 25 m and an overall length exceeding 40 m (see Fig. 1.1).

The detector is composed of individual sub-detectors, which specialize in different capacities of particle identification and measurements of signatures, plus structures for the provision of magnetic fields, cooling, mechanical support, and electronic instrumentation. The total weight of the ATLAS detector will amount
to approximately 7000 tons. Its individual sub-detectors will be stacked like layers of an onion shell around the central cavity, where provoked particle collisions occur.

As results from every collision hundreds of secondary sub-atomic particles will shoot out from the interaction point at the very core of the detector and penetrate its shells at various angles relative to the beam axis. As they pass through the surrounding material, which makes up the different sub-detectors, well-known interactions occur for particular combinations of particles and sub-detectors. Resulting effects can be measured as raw data, whose analysis allows physicists to infer about the exact nature of the registered phenomena.

Partly for practical reasons of mechanical construction sub-detectors have cylindrical rather than spherical surfaces. This fact is recorded in a nomenclature that distinguishes between barrels (the cylinders’ mantles; typically mechanically made of 2 symmetric half-barrels each) and end-caps (their roofs; the beams enter the detector cavity through their centers before collision) for most sub-detectors. The coordinate systems $\eta \times \phi$ (or $z \times \phi$) and $\phi \times R$ are used to refer to continuous points on the surface of barrels and end-caps respectively.

$\eta$ is the pseudo-rapidity: it can be envisaged as a contorted scale along the outer longitudinal margin of the detector, with $\eta \equiv -\ln \tan (\theta/2)$, where $\theta$ is the particle’s zenith angle. $z$ is the linear offset along the beam axis, $\phi$ is the angle of rotation around the beam axis, and $R$ is the radial distance from that axis.

If a sub-detector consists of well-isolated discrete elements, consecutive numbers as assigned to neighboring readout elements (such as straw numbers) are often preferred as more convenient over continuous coordinate values. As a result from the detector’s inhomogeneous shape, different detection and reconstruction algorithms have to be devised for different surface areas. The transition regions between barrels and end-caps, in particular, require special attention.

Since every detection process influences the very properties of the measured particles (such as momentum or direction of flight) in a way that must not be disregarded at the sub-atomic scale, it is important that the detector elements are arranged in the most sensible way. Stations which interact with passing particles such that these suffer only minor losses in energy and minor changes in direction (e.g. through ionization of detector gas) occupy inner layers, whereas stations that destroy particles for detecting their identity (e.g. calorimeters) and low-resolution devices must operate at outer radii. The design generally aims at reducing the overall amount of passive structures throughout the detector volume by making use of proper materials and by keeping support structures thin (as measured in units of radiation length).

By broad categorization the ATLAS detector has sub-detectors for carrying out the following kinds of observations: position measurement, energy measurement, momentum measurement, and particle identification. The third is achieved by recording several points of passage of a single particle and thus by reconstructing its approximate track. Because of the presence of a strong magnetic field the
track indirectly reveals the signed momentum of charged particles.

In the case of ATLAS the magnetic field is provided by 2 independent detector magnet systems: an inner solenoid (positioned between the TRT and electromagnetic calorimeter) and outer toroids (at about the same radius as the muon spectrometer). The first provides an axial magnetic field for the inner tracking volume (2 T at the center of the tracking volume, 2.6 T peak). The second is less usual in detector design, thus its presence contributes to the suggested abbreviation for ATLAS, as in A Toroidal LHC ApparatuS.

The overall detector optimization is guided by physics issues such as the search for super-symmetric (s) particles (some are expected already at the LEP2 energy range) and the Higgs boson (mainly expected at 1 TeV and above). While obeying the necessity to stay within cost-effective technologies, the detector should still provide as many particle signatures as possible in order to achieve robust and redundant physics measurements, because this will increase its potential for discovering new or unexpected physics.

The observable cross-sections for many of the relevant physics processes are small over a large part of the mass range to be explored by the LHC; hence the primary goal of the general-purpose experiments is to operate at high luminosity $\mathcal{L} = 10^{34}$ cm$^{-2}$s$^{-1}$ during most of their life-times in order to maximize the detectable rates above physics backgrounds. Mainly for the benefit of B-physics studies initial low luminosity running at $\mathcal{L} = 10^{33}$ cm$^{-2}$s$^{-1}$ is foreseen during the first few years of LHC operation in addition as well. This extension has important effects on the applicability of the concept of Regions of Interest (ROIs), as it is foreseen as guiding principle in designs of the LVL2 trigger system (refer to section 1.3.2).

The following chapters present the individual sub-detectors in brief views. The applied order follows their arrangement next to each other in the detector’s volume from inside out. Discriminations are based on a coarse functional view and partly on the use of different hardware technologies. Other aspects of system design emphasize different requirements and thus might suggest slightly different forms of presentation.

For instance, the extended barrel of the hadronic calorimeter can be viewed both as part of the end-caps (from a geometrical point of view: it overlaps the hadronic end-caps in z) and as part of the barrel (from a technological point of view: it is a tile rather than a LAr calorimeter). Jet ROIs (refer to section 1.3.1) are often quite large and can therefore regularly stretch across the cracks between the boundaries of barrel and end-caps. This suggests that both parts should be treated in unison, if they yield certain sets of common signatures (such as energy deposits in the calorimeter). However, this conclusion may seem artificial from other points of view (geometry, technology, readout organization).
1.2.1 Semi-Conductor Tracker (SCT)

This precision sub-detector is arranged nearest to the beam-pipe; it supports 2 aspects on layers at different radii. The inner aspect is a pixel sub-detector which provides two-dimensional spatial information on hits close to the vertex. Its granularity has been maximized by choosing the pixel sizes to be the smallest ones allowed by the area required for readout electronics.

The SCT’s outer aspect is a tracking sub-detector which attains high precision mainly in $\phi$ direction, that is only in one dimension. The approximate shape of this sub-detector forms 4 concentric cylindrical surfaces in the barrel; in the end-caps it resembles wheels that are arranged perpendicular to the beam axis at different offsets along $z$. Its internal makeup utilizes silicon double-wafers of rectangular shapes ($\sim 6 \times 12$ cm in size). The wafers carry numerous strips that are parallel to the longer edges; each strip constitutes an independent channel for particle detection. A second layer has strips that are tilted by small relative angles. The presence of strips in 2 layers introduces redundancy that partly enables measurements in 2 coordinate dimensions. In the fast-forward region technology switches to gallium arsenide substrates (GaAs), for this region is where the highest radiation doses occur.

The overall layout of the SCT is such that every track with $|\eta|$ below a certain threshold value (as determined by the masses of “interesting” particles) crosses 2 layers of pixels and 4 layers of silicon strips. It is also based on careful consideration of the partly opposing requirements for minimizing the amount of material and cost, and for maintaining an adequate number of detector channels. With regard to radiation tolerance the design is such that the detector can survive the experiment’s radiation environment during ten years of operation or more.

1.2.2 Transition Radiation Tracker (TRT)

This sub-detector consists of multiple layers of cylindrical drift tubes, whose individual straws stretch in axial and radial directions in the barrel and end-caps respectively. The intermediate gaps are filled with radiators in order to produce detectable X-ray emissions at interfaces between materials with different dielectric indices $\epsilon$. Because track densities are relatively low at outer radii, a resolution that is lower in comparison to the SCT’s (and comes at a lower per-point cost) is well sufficient.

Each hit straw contributes 2 coordinates: the first one is measured along the straw’s axis; the second coordinate is perpendicular to the straw and is available from the straw’s fixed position. Drift time information adds precision to the second. In addition to these measurements in $\phi$ and $z$, the straws of the end-cap TRT indirectly reveal $R$ as third coordinate by way of identifying the outermost straws that were crossed by a given track.

The combination of SCT and TRT forms the ATLAS inner detector. The
semi-conductor tracker in the SCT provides relatively few (and costly) high-resolution layers. The TRT's straw tubes complement this aspect by supplying a large number of continuous measurements on track trajectories (at relatively low per-point cost and while introducing less material). Both trackers are arranged such that they yield enough measurements per track for good pattern recognition, even in cases with ambiguities, such as those caused by overlapping tracks and detector inefficiencies.

1.2.3 Calorimeter

The ATLAS calorimeter consists of an inner electro-magnetic plus an outer hadronic part. Its specific aim is to reconstruct the energies and directions of electrons, photons (electro-magnetic calorimeter), and jets (electro-magnetic and hadronic calorimeters), and to measure $E_{T}^{miss}$. Barrel, end-cap, and forward regions are arranged such that they cover ranges of increasing $|\eta|$.

The electro-magnetic calorimeter and the end-cap regions of the hadronic calorimeter use a liquid ionization technique. Their passive parts consist of lead (or copper) absorber plates with liquid argon filled gaps in between. The 2 innermost barrel layers function as pre-sampler (allows for a correction of energy loss in the material before the calorimeter) and pre-shower (high granularity layer) respectively.

The design of the barrel (and extended barrel) of the hadronic calorimeter opts for a scintillator technique. This (tile) part of the calorimeter consists of staggered scintillating tiles and steel absorber material.

The forward calorimeter (electro-magnetic and hadronic parts) receives particles at the highest rapidities. It is physically integrated with the end-cap regions and occupies narrow spaces around the beam pipe. It also uses metallic absorbers plus liquid argon (LAr), but because of increased requirements on radiation hardness at higher rapidity ranges a different geometry (tubes rather than plates) and different absorber materials are used.

1.2.4 Muon spectrometer

Since it is the outermost sub-detector, the muon spectrometer defines the overall dimensions of the ATLAS detector. It also shares its overall volume with the super-conducting magnet's toroid coils.

The barrel and end-cap parts of the toroid show an eightfold symmetry around $\phi$. The muon spectrometer is built of flat and mostly rectangular chambers of one to several m$^2$ in sizes. In the barrel these are positioned at angular offsets between the structural elements of the magnet such that their combined surfaces roughly form 3 cylindrical shells of detector stations, which lie concentric around the beam-axis. In the end-caps the chambers are mounted such that they combine
to vertical planes, which attach to the inner (transition region) and outer (proper end-cap region) aspects of the end-cap toroids’ wheels.

Two different chamber technologies each are used for precision tracking and for triggering. These dualities are due to considerations of cost savings and the presence of regions which receive extremely high radiation doses. In the barrel part the ensuing design foresees Monitored Drift Tube Chambers (MDTs) and Resistive Plate Chambers (RPCs) respectively.

The MDTs consist of a central support structure, which has 2 thin multi-layers of closely packed drift tubes attached to its 2 outer surfaces. Individual tubes feature an aluminum mantle, plus a cavity that contains high-pressurized drift gas and a central wire (as is typical for various types of drift tubes). RPCs are arranged in outer multi-layers, which cover the MDT sandwiched structure.

1.3 Trigger and Data Acquisition System (DAQ)

During active runs of the ATLAS experiment bunch crossings will occur at a rate of 40 MHz, with ~ 20 proton-proton collisions resulting from each crossing at high luminosity. The detector’s overall response as it witnesses and absorbs emerging particles has to be sufficiently analyzed in order to reveal traces of new and interesting physics, whose manifestations will be superimposed on a background of particles from other (uninteresting in the context of the experiment, because known) processes.

The total data volume that emerges from the detector is ~ 1 MByte per event (~ 100 kByte of raw data per event after reconstruction [6]). High-granularity sub-detectors and a very large number of detector readout channels are required to minimize the problem of pile-up that is potentially occurring in all sub-detectors whose response time is slower than the bunch crossing period (25 ns).

The resulting challenge in data analysis has therefore been compared to the search for a needle in a haystack, with the additional complication that the haystack disappears every 25 ns for a new one [37]. (For comparison, the bunch crossing period at LEP is 22 µs [21], i.e. almost 3 orders of magnitude larger.)

In order to yield sub-tasks of lower complexity the full data analysis has been coarsely split into 2 succeeding phases: online and offline computing. They combine to form a highly selective filter. The margin between the 2 phases has been drawn at ~ 100 Hz. This rate is given by the anticipated constraints that can realistically be put onto permanent storage media, which are required for storing the remaining event data volume prior to the offline phase.

The experiment’s trigger system is synonymous with the initial phase of online computing. It has to inspect all events as they emerge from the detector in real-time and reject an overwhelming fraction in order to achieve the required rate reduction from 40 MHz down to ~ 100 Hz. This large step reduction of almost one-in-a-million between bunch crossings and recording must ensure that
the occurrence of rare events is not lost in the sea of unwanted “background” physics.

During the online phase the event data are kept in volatile RAMs. Algorithmic complexities employed at this level are relatively low, due to the required real-time behavior of the system. The technical challenge arises mainly from having to feed data into processors from distributed sources and from running algorithms at repetition rates of up to 100 kHz. (This rate corresponds to intervals of 10 μs; minimum overheads for sending single “empty” packets, as observed with some of today’s state-of-the-art communication technologies, already reach this order of magnitude: refer to chapter 3. Even higher rates occur in the LVL1 part of the trigger system: refer to section 1.3.1.) Typical algorithms presently reveal hundreds of microseconds of execution times (often excluding pre-processing), when benchmarked on today’s generation of processors.

Relevant algorithms typically operate on data sets of kilobyte sizes, and extract simple signatures by applying pattern recognition and local transformations; these entities are then compared against sets of selection criteria, thus the term trigger has been chosen for the entire device. Only events which fulfill requirements relating to critical thresholds for signature values, multiplicities, and types of trigger objects are recorded onto permanent storage media, where they await later offline analysis. The others are immediately and unrecoverably discarded from further observation.

Requirements are defined in the so called trigger menu, which is established and confirmed by testing its constituents against a large number of simulated Monte Carlo events, in order to optimize for background rejection efficiencies and rates (see Fig. 1.2 on page 12).

The emphasis of the trigger system is on a rapid quantitative reduction in data volume; this is primarily achieved by reducing the net event rate. Since the following offline phase only has to deal with the remaining subset of events, which passed the trigger system with positive confirmations (still in the order of 1 Pbyte = 10^{15} bytes per year [6]), it can focus on complementary qualitative aspects as implemented by sophisticated algorithms for reconstruction and physics analysis. Its task is to reveal the precise nature of each occurring event and to identify the underlying particle interactions.

In terms of its implementation, offline computing runs on a large farm of general-purpose workstations that is fed data from a shared pool of disks (random access, first stage) and tapes (sequential access, second stage). These storage facilities hold event data until the offline analysis catches up with previously recorded collision runs. Softwarewise the offline-system occupies the domain of large-scale databases for the handling of event data (Object Oriented Database Management Systems — ODBMS — are considered at this stage) and portable programs, which are written in high-level programming languages after abstraction from hardware. (C++ with CASE-tool support will take over from Fortran, although it is not yet clear how much Fortran-code from existing code bases will
The fact that the offline phase (unlike the preceding trigger system) reads its input data from permanent storage devices (rather than from volatile buffer memories) has practically important implications. It allows for reproducible physics results. Studies can be repeated at different times under matching conditions, or — perhaps more importantly — under varying conditions at the discretion of the physicist who performs an analysis and searches for proof of a hypothesis by applying different checks. It also permits that many collaborating institutes can share related work after partial replication of recorded event data to regional centers and home institutes, either via network connections or by movable media.

Considerations as to cost and ease of maintainance suggest that the entire filter system should be uniform and should consist of only few distinct building blocks. For instance, the use of a computing infrastructure, which can be readily bought on the market, and the adherence to the same software engineering methodologies that have been adopted as guiding principles for the construction of the offline system, should obviously govern the design of the trigger system as well.

Because of the stringent real-time processing demands that are placed on the trigger by the initial event rate of 40 MHz, this cannot be realized throughout all parts of the system. Rather the situation requires compromise, which is met in different parts of the systems in discrete steps of deviation from the stated optimum solution. The need for uncompromised speed in some parts of the system as well as the fact that not all sub-detectors’ data become promptly available for analysis at the same time lead to a design that incorporates multiple subsequent trigger levels.

The levels effectively form a pipeline, where each stage in the pipeline receives only output from the previous stage as its input. Each stage refines the selection by added complexity and takes advantage of the lower rate left over from its predecessors. Calculations at different levels overlap in time for events that were generated during subsequent bunch crossings, therefore slower sub-detectors can be tolerated by involving their data only in later steps of the analysis. As the pipeline progresses, later stages increasingly adopt characteristics of the offline system.

The ATLAS experiment uses a trigger system that consists of 3 levels. Configurations for similar HEP experiments typically vary between 2 (CMS [17]), 3 (ALEPH), and 4 (DELPHI, HERA-B, LHC-B) stages.

### 1.3.1 LVL1 trigger

The ATLAS LVL1 trigger operates on reduced-granularity data from a subset of (fast) sub-detectors: calorimetry and the trigger part of the muon-system. High-$p_T$ leptons (electrons and muons), photons, jets, and large missing transverse energies $E_T^{miss}$ are flagged as trigger objects, whose signatures are refined on
subsequent levels, by involving more data into the analysis. Physics processes that are of interest and give rise to photons and high-$p_T$ leptons include for instance the Higgs-decays $H \to \gamma\gamma$ and $H \to ZZ^* \to 4\ell$.

Fig. 1.2 on the following page shows a small portion of the trigger menu that will characterize candidate events at high luminosity $\mathcal{L} = 10^{31}$ cm$^{-2}$s$^{-1}$. The listed set of conditions reveals that LVL2 performs electron/photon identifications with higher reliabilities than LVL1 (because it uses also information from the trackers), and that it generally imposes stricter conditions over LVL1’s.

The specific task of the LVL1 trigger is to reduce the event rate from 40 MHz down to 100 kHz (75 kHz for initial running [63]). It is agreed upon that, in order to meet its requirements, the LVL1 trigger must be built using custom electronics. It executes a fixed set of highly parallel algorithms, which are re-programmable only at the level of parameters.

The LVL1 trigger operates synchronously, therefore event data is held in pipeline memories (close to the detector) while the LVL1 decision is formed. Data progresses through the pipeline with a clock period of 25 ns or a sub-multiple thereof. The LVL1 trigger starts processing of a new event every 25 ns; its overall latency is constant at $\sim 2$ $\mu$s (note that this is already considerably longer than a single bunch crossing period), with the exact final duration being determined by the number of pipeline steps and intermediate propagation delays.

The LVL1 trigger system consists of several sub-trigger processors associated with different sub-detectors plus a central trigger processor that correlates the sub-trigger results and makes the final decision. In addition to the synchronous decision (no in most, yes in only few cases), which is distributed via the timing, trigger and control distribution system, this level also provides Region of Interest (ROI) pointers for LVL2.

The concept of exploiting ROI information at LVL2 is peculiar to ATLAS. It alleviates the bandwidth requirements into the LVL2 trigger and provides a decomposition of the initial problem, because individual ROIs can be analyzed independently and in parallel.

ROI pointers are coordinates on the detector surface that identify spatially limited areas that gave rise to corresponding positive LVL1 decisions (primary ROIs), or identify locations where the detector recorded other “interesting” responses beyond lower thresholds (secondary ROIs). Electron showers in the calorimeter or tracks in the muon detector that emerge from the interaction point are examples of ROI candidates.

The LVL1 calorimeter sub-trigger processor, for instance, applies the following selection algorithm in its search for $e/\gamma$ candidates: it seeks energy clusters in 2 adjacent cells of the electro-magnetic calorimeter and requires that their energy sums are above certain thresholds. Cells as regarded by the LVL1 trigger are defined by a matrix that spans the electro-magnetic calorimeter at a reduced granularity of $\Delta\eta \times \Delta\phi = 0.1 \times 0.1$. (The full granularity, available from LVL2 onwards, varies in different regions of the detector to up to 4 times finer resolutions.)
<table>
<thead>
<tr>
<th>physics process</th>
<th>$LVL_1$ trigger</th>
<th>$LVL_2$ trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H \rightarrow \gamma \gamma$</td>
<td>$2\text{em}_{20}$</td>
<td>$2\gamma_{20}$</td>
</tr>
<tr>
<td>$H \rightarrow ZZ^* \rightarrow 4\ell$</td>
<td>$2\text{em}<em>{20}$, $2\mu</em>{6}$, $1\text{em}<em>{30}$, $1\mu</em>{20}$</td>
<td>$2e_{20}$, $2\mu_{6}I$, $2e_{30}$, $1\mu_{20}I$</td>
</tr>
</tbody>
</table>

Figure 1.2: some trigger menu items at high luminosity (reproduced from [5])

<table>
<thead>
<tr>
<th>sub-detector</th>
<th>no. of channels</th>
<th>no. of ROBs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SCT$</td>
<td>$6.2 \times 10^6$</td>
<td>256</td>
</tr>
<tr>
<td>full barrel</td>
<td>$3.2 \times 10^6$</td>
<td>132</td>
</tr>
<tr>
<td>one end-cap</td>
<td>$1.5 \times 10^6$</td>
<td>62</td>
</tr>
<tr>
<td>$TRT$</td>
<td>$424.4 \times 10^3$</td>
<td>512</td>
</tr>
<tr>
<td>half barrel</td>
<td>$52.5 \times 10^3$</td>
<td>64</td>
</tr>
<tr>
<td>one end-cap</td>
<td>$159.7 \times 10^3$</td>
<td>192</td>
</tr>
<tr>
<td>electro-magnetic calorimeter</td>
<td>$189.4 \times 10^3$</td>
<td>432</td>
</tr>
<tr>
<td>half barrel</td>
<td>$53.8 \times 10^3$</td>
<td>112</td>
</tr>
<tr>
<td>one end-cap, $</td>
<td>\eta</td>
<td>&lt; 2.5$</td>
</tr>
<tr>
<td>one end-cap, $</td>
<td>\eta</td>
<td>&gt; 2.5$</td>
</tr>
<tr>
<td>hadronic calorimeter</td>
<td>$14.6 \times 10^3$</td>
<td>48</td>
</tr>
<tr>
<td>half barrel</td>
<td>$5.1 \times 10^3$</td>
<td>16</td>
</tr>
<tr>
<td>one end-cap</td>
<td>$2.2 \times 10^3$</td>
<td>8</td>
</tr>
<tr>
<td>muon system</td>
<td></td>
<td>208</td>
</tr>
<tr>
<td>total RPCs</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>total MDTs</td>
<td></td>
<td>192</td>
</tr>
</tbody>
</table>

Figure 1.3: detector readout parameters (incomplete, reproduced from [10])
An additional threshold of 1 GeV is applied locally to all cells before including them into sums, in order to reduce the effects of electronic noise and pile-up. All following calculations are done in integer arithmetic in units of 1 GeV. Considering 2 trigger cells at once helps in avoiding inefficiencies when showers are shared between adjacent cells. Additional thresholds for imposing upper limits on the combined energy in the 12 cells that form a ring around the cluster (isolation criterium) and in all cells of the hadronic calorimeter that are intersected by their projection (hadronic isolation criterium) can be used to veto any cluster and reject much of the jet background to the e/γ-triggers. (Jet-production will be the dominant high-p_T process at the LHC.)

The LVL1 jet trigger algorithm similarly inspects windows of Δη × Δφ = 0.8 × 0.8 in both the electro-magnetic and hadronic calorimeters. (Jets are a mixture of electro-magnetic and hadronic energies.) Windows are aligned with their upper left corners along a grid with Δη × Δφ = 0.4 × 0.4, such that closest windows overlap in both dimensions by Δη = Δφ = 0.4. The employed algorithm effectively considers windows that shift by small offsets.

### 1.3.2 LVL2 trigger

The ATLAS LVL2 trigger is guided by ROI information that is passed on to it from LVL1. It uses full-granularity data from most sub-detectors, but considers only detector regions in the vicinity of ROIs. The number of flagged ROIs per event is estimated as ~ 5, on average, for high luminosity runs.

The size of ROIs varies between sub-detectors. Overall, the LVL2 system needs to include only data from ~ 10% of all ROBs in its analysis. If raw data can be selected from ROBs on a lower granularity than entire buffers, this percentage shrinks further (~ 1% in [21]).

The principle of ROI guidance can offer only limited steering during low luminosity runs, where other triggering strategies must be applied. The foreseen LVL1 trigger for B-physics studies is now based on a single muon above 6 GeV, followed by sequential processing on LVL2. This constitutes one of the main reasons why the trigger system should be kept highly flexible and reconfigurable (also on the level of algorithms) from LVL2 onwards.

The specific task of the LVL2 trigger is to reduce the event rate from 100 kHz further down to ~ 1 – 5 kHz, a rate that can be sustained by the projected event building system. Unlike LVL1, it performs asynchronous operations on events, with event decision latencies of up to few milliseconds.

Unlike at LVL1, where a pipeline is foreseen for aggregating processing such that execution times exceeding the periods between arrivals of new events can be tolerated (synchronous operation), some of the proposed implementations for the LVL2 trigger foresee a “horizontal” (rather than “vertical”) aggregation of processing elements into farms. Once a processing element in a farm is assigned a work load under this scheme, it stays on the task until finished. This operates
based on the assumption that a sufficiently large total number of processing elements is available, such that idle ones can take over as equal substitutes, while others are still busy (asynchronous operation). Part of the reason for this reorganization is that algorithms at LVL2 don’t lend themselves well to decomposition into a linear and unchanging sequence of steps, as can be identified for the simpler algorithms at LVL1.

The necessity to provide an agent that distributes work between the processing elements of LVL2 (supervisor, itself becoming an obvious hot-spot of the system) and the requirement for a communication network for the distribution of workloads between and gathering of results from a large number of peers within narrow time constraints present the main challenges to the implementation of LVL2. Technology limits may impose changes to its present outline design.

### 1.3.3 LVL3 trigger (event filter)

The ATLAS LVL3 trigger (event filter) carries on from LVL2’s function and attempts a further reduction of event rates from \( \sim 1 \) kHz down to \( \sim 100 \) Hz on average. Depending on the type of event, data volume reduction is achieved by a combination of event selection and possibly event compression.

The LVL3 trigger considers full-granularity, full-resolution data from all subdetectors in its analysis. After having formed a decision, it can optionally shrink data for most standard events, while reserving the full 1 MByte/event bandwidth for others (such as Higgs boson candidate events).

To the extent that an implementation is compatible with this level’s projected event decision latency of \( \lesssim 1 \) s, LVL3 already closely resembles the following offline stage. Indeed, an attempt will be made to reuse offline code in this phase.

The LVL3 trigger receives its input from the event builder. This device has the task of gathering data fragments that belong to the same event from buffers that are dispersed, as the detector’s readout organization dictates, into uniform buffers, which sit close to the processor for carrying out LVL3 processing of the same event. (Its function roughly resembles all-to-all communication in MPI parlance, as illustrated in Fig. 3.6 on page 76.)

Up to the current generation of LEP experiments buses such as CAMAC and VME have been used to establish the required interconnectivity between multitudes of buffers and processors. Because of the considerable growth that will occur with the event of LHC detectors and the unsatisfactory scaling properties of buses (due to increasing congestion on a single shared medium), switching technologies such as SCI, ATM (refer to section 3.3.2), HIPPI switches, or pragmatic extensions of popular media, such as Fast Ethernet or Gigabit Ethernet, are foreseen for use in future event builders [69], in addition to their possible use as communication media in farms as well.

The implementation of the LVL3 trigger uses parallelism on the event level; once a given processor has been assigned an event and has received its full data
via the event builder, it requires no further communication with other nodes. Therefore a farm of general-purpose workstations, i.e. a loosely coupled parallel system, is well suited for the LVL3 task.

Unlike on LVL1 (where it is not an issue) and on LVL2 (where it is still disputed), the use of operating systems on individual nodes is a firmly established must for LVL3. UNIX and some of its real-time-oriented flavors are considered for final use and typically preferred also during prototyping efforts. Windows NT appears to many groups as a potential vehicle for enabling low-cost implementations based on mass-market PCs. As in many other related research efforts, it is currently preferred over Linux; the latter nevertheless remains the declared favorite of many grassroot efforts.

1.3.4 Implementation choices for the 100 kHz trigger

LVL2 sits in between 2 adjacent levels also in terms that translate very literally into its design. It is not yet clear whether an implementation can afford to opt for the use of commercial off-the-shelf components, such as general-purpose workstations (LVL3), or whether it will have to be built along the more stringent lines of LVL1. Much depends on whether one shares a conservative or optimistic view as to the further continuation of technology advancements until the time when the trigger system will be deployed with the ATLAS detector.

The applicability of Moore’s law (the increase of processor performance by a constant factor after regular time intervals — about 18 months for a factor of 2 at present) is rarely refused; nevertheless engineers are less comfortable in projecting the future evolution of high-speed communication networks and processor interfaces in quantitative terms.

An evolution nearing the performance curve of processors cannot be predicted with much confidence for these technologies. In determining past growth one cannot look back at a history as long as the one that forms the empirical background for Moore’s law as it applies to microprocessors. Many emerging and promising technologies are only starting to make their appearances outside labs as of now. While e.g. ATM (refer to section 3.3.2) aims at homogeneously covering the needs of both LANs and WANs, and therefore has the potential of achieving sufficiently wide-spread use for driving prices down, sufficiently large switches are at present solely used by telecommunication companies in their backbones and are therefore sold in peculiar price envelopes.

It is also unclear to what extent the telecommunication market’s (as the strongest driving force’s) end-user applications, which are mainly hungry for bandwidth, will cover the needs of the ATLAS LVL2 trigger well. Its performance prediction is mainly influenced by network latency, due to the sustained passing of numerous small messages. (There are obviously some cases where latency also matters more in telecommunication, such as two-way digital video, for instance.)
Figure 1.4: effects of partitioning tasks on overall required effort

As for parallel computers, prices of the largest systems (such as the Cray T3E, which scales up to 1024 nodes) will perhaps remain prohibitive, and many of them will be restricted by the amount of “openness” in their outside connectivities. It may nevertheless become feasible to reuse technologies which have been developed in a similar context, such as e.g. SCI (refer to section 3.3.4).

Fig. 1.4 suggests a metaphor that illustrates why low-latency communication is important to our application. It is quoted from [13], where it refers to communication in a team of human professionals.

Fig. 1.4(a) and Fig. 1.4(b) represent the extreme cases of perfectly partitionable tasks (completely independent sub-tasks can be identified, and each sub-task requires exactly the same amount of effort) and unpertitionable tasks (the time until completion $y$ cannot be shortened by involving more personnel $x$). The author observes that partitionable tasks incur new overheads, because team members cannot always work exclusively on their sub-tasks, but must allow
for communication in order to coordinate and exchange input and results. The example that is given in Fig. 1.4(c) assumes individual discussions between all team members, thus \( y = c_1 \frac{1}{x} + c_2 \frac{x(x-1)}{2} \). \( c_1 \) indicates the required effort if one worker attacks the task alone, \( c_2 \) parameterizes the cost of communication.

As can be seen from the ascending tail in Fig. 1.4(c), this highly “democratic” team can even end up spending more time on the task than an individual, depending on the team’s entire size. The concept of performing sub-tasks in parallel can be rendered meaningless with regard to the team’s overall efficiency, if too much time is spent on communication. (The second term in the previous formula can obviously also be reduced by introducing a command hierarchy, in order to avoid costly all-to-all communication.)

When applying the lesson to the task of the LVL2 trigger, we can observe that — even if the problem allows for a decomposition into sub-tasks — cost for individual communications must be kept low. For small messages this cost is defined by the network latency (refer to chapter 3), with point-to-point bandwidths gaining in relative importance only as message sizes increase.

Different appraisals of the relative importances of related considerations (and a wide range of free-flowing research interests) have led ATLAS collaborators to pursue 3 different solutions up to the development stage where demonstrator systems were put to test in controlled lab environments.

The 3 approaches are termed architecture A, B, and C respectively. Although they should present mainly a variation in architectural choices, their current manifestations are also coupled to specific technological choices, for reasons that are partly arbitrary and historical. This will make it more difficult to disentangle effects of mixing architecture and technology studies, when the best architecture for the LVL2 trigger will have to be chosen. By present understanding, each architecture is potentially capable of implementing all required functional steps.

Architecture A and B appeared already in the outline that was given in the ATLAS Technical Proposal [5], where they were described as data-driven and farm-based solutions respectively. Architecture C was added later as specifically optimized approach towards a sequential processing strategy, where the tasks of LVL2 and LVL3 can merge into one system.

1.3.5 Building blocks of the LVL2 trigger

Where a generic description cannot be achieved, the following presentations make implicit references to architecture B (refer to section 1.3.7), because it incorporates the largest count of discrete building blocks.

1.3.5.1 Supervisor

The supervisor receives positive LVL1 decisions plus related ROI pointer information from the LVL1 trigger. It serves as a gateway for control information into
and out of the LVL2 trigger system. Its task is to distribute work for efficient use of the parallel system, such that redundant resources are kept busy by equal measures. With properly balanced occupancies, speedup through parallelism enables the entire system to keep up with the overall event rate, although the aggregated computational delays per event remain markedly longer than those given by the desired global repetition rate of 100 kHz.

In order to accomplish its scheduling task, the supervisor has to maintain up-to-date status information on all components. The device makes its information partly available for use by outside agents, like the detector control system (DCS), which uses it to monitor the physics performance and hardware performance of the detector. This function is clearly of great importance to the experiment, because all data that is rejected by any of the 3 trigger levels cannot be recovered and is therefore lost for later offline analysis.

The supervisor also monitors the operational status of the LVL2 system. It detects failing components by setting timeouts for all requests. When a component has not replied to a request after the timeout period has passed, the supervisor may restart it autonomously, if the hardware configuration allows so. Failure in any part of the system usually leads to acceptance of the processed event; this leads to the confinement of errors on the conservative side. Repeated errors of the same device lead to its exclusion from future scheduling decisions.

The supervisor uses collected status information also to throttle or inhibit input from the LVL1 trigger in case that its internal buffers near saturation. Similarly, it accepts throttle signals from the downstream LVL3 trigger. (The mechanism is meant for application in rare cases of exceptional loads.)

The supervisor has to tolerate the full 100 kHz event rate at all times. This demand and the desirability of its increased fault tolerance through provision of multiple identical resources suggest that it should itself be implemented as a parallel computer. A specific design, which was used with modifications in all architectural demonstrations, foresees several (up to around ten) embedded RISC-processors in a VME crate, plus a high-speed proprietary bus that accomplishes the initial attribution of work to one particular processor.

### 1.3.5.2 Readout buffers

At about the same time when the supervisor receives information from LVL1, event fragment data arrive from the sub-detectors' front-ends via Readout Drivers (RODs) into the Readout Buffers (ROBs). The ROBs must hold all data for events whose LVL2 decisions are still pending. Their depth (size) is therefore a direct impact of the LVL2 overall latency.

Depending on the LVL2 decision, ROBs either discard data for an event from their memories, or send it on towards the event builder and LVL3.

The overall number of ROBs for ATLAS is estimated at ~ 1500, which leads to an event fragment size of ~ 1 kByte per buffer and per event. This buffer
count was established based on the assumption that data arrives on individual optical fibers that are driven by RODs and carry bandwidths of 100 MByte/s each (1 kByte × 100 kHz = 100 MByte/s). Fig. 1.3 on page 12 provides a more detailed listing of detector readout parameters, in numbers of readout channels and numbers of ROBs for different sub-detectors, according to present understanding.

RODs have the purpose of multiplexing many (millions of) low-bandwidth detector readout channels onto fewer (high-bandwidth) optical lines. They also constitute a barrier, from which onward uniformity throughout the entire extent of the detector’s readout organization is highly desirable. In contrast to this, elements that are further upstream must be allowed to adopt each sub-detector’s local and divergent conventions, out of pragmatic necessity.

Apart from input from LVL1 (via RODs) and output to LVL3 (via the event builder), RObS also duplicate parts of their contents and push them into the LVL2 trigger system, whenever processing of a new event is initiated by the LVL2 supervisor. The supervisor enables event-parallelism in the global system by assigning one of its idle processors to each subsequent event. As for the local system, the assignment between processors and ROIs is either also done by the supervisor in an analogous way, or it follows a static scheme of “geographic” allocation, whereby each processor is permanently attached to ROIs from a certain region of detector surface.

There are 2 options for the supervisor in relaying control information: it may either broadcast ROI information for an event along with the identification of the chosen processor(s) to all RObS, in which case RObS have to determine whether they contain part of the data at the full rate of 100 kHz, and if so send it to the concerned processor in the local system. Alternatively, the supervisor may acquire the task of selecting concerned RObS on behalf of the multitude of all RObS. In this case it sends ROI information only to specific targets on point-to-point links. Although sending \( n \) messages is up to \( n \) times more expensive than sending a single broadcast message, the scheme obviously relieves individual RObS from having to perform buffer management operations at a 100 kHz rate (at the price of putting even more load on the supervisor). The new net rate for given RObS is lower, because ROIs positions change for subsequent events.

Another potential challenge that is directly related to the 100 kHz rate exists, because RObS have to accept LVL2 decisions, as broadcast from the supervisor, and act accordingly in their buffer management. (ROBs must not ignore this information, for otherwise their buffers would soon overflow.) This rate can be brought down by grouping decision transfers.

A variant scheme foresees staged communication between the supervisor and RObS via ROI distributors. Point-to-point messaging and broadcasting can be intermixed between the 2 stages as specific technologies permit. The natural granularity for the assignment between RObS and ROI distributors is one distributor per crate.
1.3.5.3 Feature extraction processors

The application of physics-related algorithms to a particular ROI commences in parallel in processors in several local systems, as determined by the supervisor from ROI pointer information. Calculations at this stage pertaining to a given event are collectively referred to as feature extraction (FEX).

Prior to this, a separate processing step (not yet physics-related) can optionally prepare raw data when they arrive in detector-dictated formats and reformat them such as to meet specific requirements of LVL2 algorithms. This pre-processing may include manipulations such as decompression, rearrangements, and rescaling.

Two partially opposed considerations suggest that pre-processing should be treated (at least conceptually) as a separate task: the task may consist of operations for which the local system's general-purpose processors are unsuitable (e.g. bit field operations); thus it may profit from hardware with separate competence (refer to section 1.3.9). On the other hand, current ROB implementations include microprocessors, for accomplishing memory management functions, in their designs. These flexible devices can potentially be occupied with responsibilities relating to pre-processing as well.

The exact nature of pre-processing still needs to be determined, as sub-detector designs evolve. Until then the abstract concept serves as a place-holder for physical implementation choices, which can be introduced at different relative offsets between ROBs and feature extraction processors.

1.3.5.4 Global processing

During feature extraction each processor attempts to do as much decision preparation as it possibly can, based on the ROI information that it has locally available. It parameterizes the contents of the ROI by calculating a small set of features. Typical features describe energy peaks or track parameters, for instance. All FEX processors leave the final LVL2 decision to the assigned processor in the global system, which inspects features rather than raw data, and compares them to multiple physics interpretations (hypotheses). Features are variables that allow improvements in the signal/noise ratio by contributing to the rejection of events that are defined as background, while retaining the highest possible fraction of interesting physics events.

ROI collection refers to the operation of assembling information from all ROBs (in a sub-detector) that are needed to process a given ROI, and possibly rearranging this information in a format and order that serves to facilitate feature extraction. In architecture B (refer to section 1.3.7) this functionality is implemented in software. Architecture A (refer to section 1.3.6) uses a special-built interconnect, called router, instead. The router acquires the role of the network in the local system.
The final LVL2 decision on an event (global decision) can e.g. involve the combination of track segments from different inner detectors and match them with clusters in the pre-shower and calorimeters. It consists of 2 steps: first the features from different sub-systems are combined to produce the global features for each ROI; then the features from the different ROIs are combined for the final decision. It is at this stage that the trigger system first combines evidence from several sub-detectors to form its intermediate decision.

The LVL2 decision propagates back to the supervisor, along with topical status information from processors in the local and global systems. (The loop segment from local processors to the supervisor traverses the global system.) The supervisor in effect authorizes ROBs to remove data fragments for the concerned event from their memories, and updates its status view on processors that have become available for the assignment of new tasks (if applicable).

### 1.3.6 Architecture A

Architecture A is the agreed-upon fall-back solution for the implementation of a critical part of the LVL2 trigger. Its preparation covers a possible scenario where key technologies cannot deliver in terms of their future evolution towards meeting ATLAS’ challenging requirements for operating at 100 kHz. Very much like LVL1 in spirit, architecture A deliberately sacrifices reconfigurability for top speed.

Its technological substrate are Field Programmable Gate Arrays (FPGAs), whose programming occurs at the gate-level. They require more substantial effort (description languages such as VHDL are specific to the task and offer only little abstraction; turn-around times during software development can be in the range of hours) and have to obey unfamiliar limitations (support for floating-point operations imposes a hard challenge; algorithms must be tailored to a specific hardware setup) in comparison to general-purpose processors of the type that serve e.g. in workstations.

However, depending on what mainstream technology can offer at the time when buying decisions for ATLAS will have to be made, these obstacles may be rendered unimportant in light of the architecture’s specific advantages. It has been successfully demonstrated that a combination of several FPGA processors on a VME board (the Enable++ system [42]) can perform feature extraction for a number of sub-detectors, while fully keeping up with the 100 kHz event rate required by LVL2.

Therefore an aggregation of as many Enable++ boards as is suggested by the maximum number of ROIs per event (~ 5 on average: note however, that this stems from a distribution with a tail; less than 1% of events have more than 10 ROIs), can accomplish the task of feature extraction for all sub-detectors in synchronous operation. Each ROI generates a separate data stream into one pipelined processor.
1.3.7 Architecture B

Architecture B is an implementation of the baseline design that was introduced in the ATLAS Technical Proposal [5]. It strives to make maximum use of the double-parallelism (sub-detectors and ROIs) that is inherent in the task of the ATLAS LVL2 trigger, by assigning independent workloads to sub-systems (farms and processors) that are physically separate and that can therefore work in parallel. This straightforward mapping leads to as many farms as sub-detectors need consideration (local systems) plus one more farm for combining their results (global system). The required communication networks may actually form partitions of one physical network.

Architecture B is “in principle” also committed to the use of general-purpose processors, which have the backing of large market segments outside HEP, and commercially available networking technologies, for providing connectivity between data sources and processors.

While the principle always holds at the chip-level, and much tribute was paid to it in describing the positive spin-off effects in terms of the affordability, manageability, upgradeability, and cost control of a system that was built along its lines, it is not clear where the margin that bounds commercial systems shall be drawn in real life. (Whatever components make it into the final ATLAS LVL2 trigger system will be commercial at least in some sense, because the size of the detector requires production in industrial settings.) Implementers working on architecture B had to invest large efforts in constructing boards, interface cards, software libraries, and tools. To the extent that emerging industry standards were adopted (such as in the case of most SCI activities, for instance) these designs may themselves develop into commercially available components, or may eventually be replaced as manufacturers of compatible equipment take over.

Nevertheless most of what went into the relevant demonstrator program was home-built from commercially available chips and components — but so was architecture A, whose current manifestation performs much faster in comparison. However, architecture B as such has a higher potential for absorbing mass market developments into its design: its local and global systems are conceptually farms of general-purpose processors with one centralized supervisor each, for distributing work between all nodes. (Where Digital Signal Processors were used as nodes, implementations did not depend on their special-purpose processing capabilities; instead, they used them as a means for partially integrating the network interface on the chip-level, by making use of their transputer-like communication lines — refer to section 2.2.1).

The physical implementation maps the parallelism of ROIs from different sub-detectors onto resources in several local systems. The secondary parallelism occurs from the existence of several ROIs per event; it maps onto different processors in the corresponding local systems (local relative to each sub-detector).

The pursued demonstrator programs adopted the following specific technology
choices so far:

- DSP320C40 Digital Signal Processors [72] and Alpha-based workstations as processors in the local and global systems respectively, and a network of C104 Asynchronous Packet Switches [68] and SCI [47] as corresponding communication fabrics

- Transalpha workstations (Alpha RISC-processors with transputers as their communication coprocessors) and identical configurations for the global system and networks (architecture B1)

- RIO\(^1\) embedded processors (PowerPC-based system on a single VME board) and SCI in the local system plus the same global system as above (architecture B2)

Unlike in architecture C, architecture B requires only unidirectional communication, although some media may voluntarily support bidirectional traffic. In particular, all paths of control information in architecture B follow unidirectional closed loops, where both requests from senders and feedback from receivers complete cycles and pass by the supervisor. This presents the supervisor with the opportunity to mastermind the system’s entire control, but also puts a heavy burden on the physical device.

Since event fragment data in particular are pushed by ROBs into the local system after the supervisor has sent commands to these ROBs, the (default) mode of operation of architecture B is called push mode, as opposed to architecture C’s pull mode. It permits that ROBs send their data only after explicit requests from receivers. (The latter relies on the passing of backward-bound control information, and therefore on bidirectional communication.)

The push mode of architecture B is in particular optimized for memory-mapped communication interfaces, where senders can write directly into receivers’ memories, as it is the case for instance with VME [38] or SCI (refer to section 3.3.4). In its most literal version, this avoids extra memory-to-memory copies and eliminates the need for flow control as well as local checks for buffer overflow (refer to section 3.1.1). The scheme requires that receivers pre-allocate circular shared buffers; senders address them by using event numbers modulo the maximum number of pending communications (as enforced by the supervisor) as proper indices.

Architecture B simplifies local communication by centralizing all flow control at the supervisor. It deliberately tolerates the introduction of slight inefficiencies relating to memory and CPU occupancies, as caused by varying distributions in packet sizes, (receivers have to pre-allocate all buffers at maximum sizes) and

\(^1\)http://www.ces.ch
propagation delays in the networks (for instance, some time passes before the supervisor learns that a processor went idle and is ready for reassignment). Therefore the final system has to be over-specified by applying pessimistic evaluations of the required buffer space and CPU capacities, with the exact quantities being ideally revealed by modeling exercises (refer to chapter 4).

1.3.8 Architecture C

The characteristic feature of architecture C is its reliance on a (physically and logically) single farm of processors. The ROI-level parallelism that is fundamental to architecture B is neglected in architecture C (as far as a direct mapping onto the physical implementation is concerned), in favor of enabling a model for sequential processing of the trigger algorithms.

One processor in the farm carries out the full LVL2 processing for a particular event. The processor executes algorithms from the trigger menu in a sequence that is optimized towards achieving efficient early cuts in event rates. As soon as one condition fails, a no-decision can be generated for the event without more algorithm executions needed.

The requirement for sequential processing has already been firmly identified for the trigger menu that serves B-physics studies during low luminosity runs [15]. Another part of the motivation for considering a sequential processing strategy for the high luminosity trigger menu comes from the fact that some feature extraction algorithms (tracking algorithms for the inner detector) are slower than others by a factor of $\sim 10$.

Although architecture B can be forced into sequential processing mode by manipulating the supervisor’s scheduling strategy, the attempt is somewhat artificial in light of the architecture’s original claim for parallelism, and it wastes redundant resources as idle. Architecture C accommodates B-physics triggers at low luminosities more naturally and allows for an implementation that is conceptually simple and homogeneous (built of few types of discrete elements). In addition, it relieves the supervisor from some of its periodically reoccurring tasks, thus reducing the likelihood of making it the system’s major bottleneck.

Processors in architecture C use pull mode rather than push mode for acquiring event data fragments. In this mode ROBs essentially act as passive devices and do not generate traffic into the farm, unless specific requests arrive from downstream processors. This “just-in-time” delivery of data is in direct relationship with the processors’ sequential processing strategy: algorithm steps whose contributions to the LVL2 decision have already been rendered obsolete by previous conditions are no longer executed, nor are related communications. Events can be rejected after any processing step which reveals that they are no longer in agreement with any set of trigger conditions from the trigger menu.

Architecture C’s separation into processing steps is a functional concept, and does not imply a physical correspondence as in architecture B. Since the natural
Locality of algorithms is by sub-detectors, processors will typically request ROI data from isolated sub-detectors during early sequential steps. Processors could nevertheless continue in their acquisition of event fragments past the size of all ROIs, until they possess full data for each event. In this case architecture C's switching network acts as a full event builder, and the same hardware performs the tasks of both LVL2 and LVL3 in a single computer system [14].

Some potential drawbacks that are specific to architecture C have been identified: in order not to waste resources while requests to ROBs are pending, individual farm processors will have to work on several events concurrently. The resulting context switching times between threads add to the latency of the LVL2 system. Farm processors in architecture B can operate in a comparatively simpler mode, by polling queued work loads from their input buffers (push mode).

In addition, architecture C requires full connectivity between all ROBs and farm processors, for allowing its processors to acquire data from all sub-detectors without forwarding from (now absent) intermediate local systems. This all-to-all connectivity substantially increases the overall size of the switching network, as compared to architecture B's local and global farms. Switch-to-farm interfaces (SFIs) have been proposed in order to make efficient use of high-bandwidth ports through multiplexing, and thus to reduce the overall number of required ports.

1.3.9 Quantitative comparison between readout formats for the end-cap TRT

For sub-detectors whose occupancies can be estimated as low ($\ll 100\%$) the readout organization clearly benefits from protocols which foresee some kind of data compression on the communication lines between the readout electronics for shaping, amplifying, and digitizing signals and the ROB memories that form the gateway into the asynchronous part of the detector's trigger system (LVL2 and downstream). For otherwise the high rate of synchronous transmissions ($\sim 100$ kHz upstream from the LVL2 trigger) and the large overall number of readout channels would require the installation of expensive high-bandwidth transmission media, such as optical fibers, in prohibitive quantities. If a proper level of data compression can be achieved, the transmission lines and attached communication buffers can be implemented at lower cost through the efficient use of multiplexers.

Compression on-the-fly (compression that occurs with no cost other than transmission time) would be most desirable from the point of view of requirements from other attached components, because otherwise their new demands for additional buffer memories may quickly eat up any savings that were gained through the use of compression in the first place. This can hardly be achieved if the tasks of compression and decompression are left to those general-purpose microprocessors, which deal with other aspects of the communication protocols (such as routing) or physics-related computing. These processors are usually ill-
equipped for performing the required bit-level manipulations, yet the provision of special-hardware components in the data-stream incurs new cost. In the TRT sub-detector the choice for data compression has already been firmly established, yet at the time when this study was performed the specifics of the exact readout format were not yet fixed.

Fig. 1.5 shows a schematic view of one end-cap’s outer surface after its projection into 2 dimensions. Due to this projection straws appear as “dots” in the grey areas. This geometry is based on the detector description as of March 1995. One can identify 18 vertical wheels along $z$, or $152 + 72$ parallel planes on a higher level of granularity. Planes are equidistant in $z$, whereas the wheels are not. Wheels fall into 2 categories: large and small wheels, which hold 16 and 8 planes respectively. They can also be subdivided into 192 modules along $\phi$, where each modules stretches across the total longitudinal extent $\Delta z$ of all wheels. The inner 13 wheels’ modules intersect 4 straws per plane, in the 5 outer wheels this number is only 3. Thus the total number of radial straws per plane is $4 \times 192$ for the inner wheels and $3 \times 192$ for the outer wheels.

This subdivision is a direct manifestation of the readout elements’ physical mounting; thus corresponding indices for identification of particular modules, planes, and straws, as provided by local readout electronics, appear as “coordinate values” in several suggested readout formats, namely in those which use zero suppression.

For the sake of brevity in the following discussion the numbers of modules, planes per module, and straws per plane are abbreviated as $n_1$, $n_2$, and $n_3$; their proper values for one half of the detector are 192, $152 + 72$, and 4 respectively.

Figure 1.5: $\eta \times \phi$ cut-away view of the “unrolled” end-cap TRT
(The latter ignores the small discrepancy of 1 between the number of straws per plane in the case of inner versus outer wheels, and induces slightly pessimistic estimates for $b_i$ in the following.) Furthermore, $s_i = \lfloor \log_2 n_i \rfloor$ is the number of bits required for storing the relevant ranges of values, and $n \leq \prod_{i=1}^{3} n_i$ is the total number of straws.

The following functions are required as well: $p_3(p)$ is the probability for a straw to be hit provided that $m_1(p)$ modules plus $m_2(p)$ planes do not contain any straws that were hit. The formulas for $m_1(p)$ and $m_2(p)$, which are given below, assume that hits follow an equal distribution in $z \times \phi$.

$$p_3(p) = \frac{n_1n_2p}{n_1n_2 - n_2m_1(p) - m_2(p)}$$

$$m_1(p) = n_1(1 - p)^{n_2n_3} \quad m_2(p) = n_2 \left(1 - \frac{n_1p}{n_1 - m_1(p)}\right)^{n_3}$$

$p$ is the occupancy of the end-cap TRT, i.e. the relative frequency of straws that report non-default (non-zero) values to their readout channels. Occupancies for this particular sub-detector are estimated at $\sim 5 - 10\%$. For simplicity, the following calculations assume that hits are randomly distributed across the grey areas in Fig. 1.5. This is hardly realistic and almost contradictory to the concept of ROIs (locally confined accumulations of manifest detector response). Fig. 2.2 on page 35 shows a realistic sample event in a prototype of an end-cap TRT under test-beam conditions. While formats without zero suppression are not affected by an erroneous description of the shape of hit distributions, formats with zero suppression in reality gain from the likely absence of hits in entire blocks, therefore results for these formats must be treated as conservative estimates.

Fig. 1.6 on the following page presents the selection of readout formats that was considered in the present study. Data from each straw were represented by 15-bit vectors, with identical interpretations in all formats. The bits conveyed information from 3 subsequent bunch crossings in an encoding that is illustrated in the upper left corner of Fig. 1.7. No attempts were made at the time to apply compression also at the level of fields inside these bit-vectors, although the upper right corner of Fig. 1.7 sketches a possible scenario.

The following benchmarks operate on the assumption that data arrive in any of 5 possible input formats that are briefly described in the following, and that an algorithm for pre-processing has to convert them into the output format that is shown in the lower half of Fig. 1.7. This output format has been chosen precisely to suit the assumptions of an existing implementation of a realistic feature extraction algorithm for the end-cap TRT. (Refer to chapter 2.1 for more information of a version of the algorithm that was simplified and tailored for test-beam runs with a prototype of the end-cap TRT.) The fundamental difference
Figure 1.6: candidate readout formats for the end-cap TRT

Figure 1.7: output formats before feature extraction in the end-cap TRT
in encoding between input and output formats is that the first (in uncompressed versions) describe bitmaps, whereas the second uses flat lists of hits’ $\phi$ and $z$ coordinates. The following details relate to the 5 input formats:

**Format 1** uses no compression at all, thus each existing straw contributes 15 bits of information, regardless of whether it was hit or not (bitmap).

**Format 2** is almost equivalent to the previous one, but trades in one extra redundant bit per straw for the obedience of byte boundaries between individual fields. This typically facilitates (and speeds up) data handling in code that has been developed for general-purpose microprocessors (aligned bitmap).

**Format 3** uses a probabilistic code, where the most frequently occurring bit pattern (for low occupancies it is obviously the one which represents non-hit straws) is replaced by a reserved sequence of bits, which has the particular advantage of being short (zero suppression). Only 3 out of 4 two-bit patterns are required for encoding valid pulse heights (0, 1, and 2) in the first 2 bits of each 15-bit field. The idle pattern can therefore serve as abbreviation, if 2 bits are deemed as short enough replacement. Alternatively, yet another format could foresee a one-bit abbreviation, but then all 15-bit fields for hit straws would have to be extended by 1 bit in order to make the necessary distinction between unabbreviated and short patterns.

**Format 4** uses a layered zero suppression encoding. Only information about hit straws is included in the representation. The 15-bit field for every hit straw is preceded by the straw’s coordinates, as expressed by a vector consisting of module number, plane number within the module, and straw number within the plane. The layering foresees that all straws that have matching coordinates in the lower dimensions of their vectors are put together in a block (the order of straws no longer matters, if each straw carries full coordinate information), and that the first coordinate is given only once at the beginning of the block, thus allowing for a more efficient representation (with one coordinate less) within. The required data volume (in bits) per entire half-detector for this format is given by the equation:

\[
b_1(p) = s_1 + (n_1 - m_1(p)) \left( s_1 + s_2 + (n_2 - m_2(p)) \left( s_2 + s_3 + n_3 (s + s_3) p_3(p) \right) \right)
\]

**Format 5** is again almost equivalent to the previous one, but, in addition, aims at obeying byte boundaries between individual fields. It represents another choice of compromise between minimal bandwidth and ease of handling in general-purpose microprocessors.
Fig. 1.8(a) compares the values of $b_i(p), 1 \leq i \leq 5$, for each of the 5 formats. Obviously input formats with zero suppression (especially those with layered zero suppression) present themselves as attractive choices if small data volumes at low occupancies prevail. Their amount of additional header information does not exceed what is gained by compression until up to occupancies of $\leq 50\%$.

In order to put these observations into perspective, conversion algorithms were implemented for all 5 combinations of input and output formats and executed on a Sun SPARCstation 5 (85 MHz microSPARC-II CPU, rated at 64.4 SPECint92 and 53.1 SPECfp92). This exercise related to a more general discussion in ATLAS about the suitability of general-purpose processors for specialized low-level tasks and used a rather bold example for illustration.

All algorithms were implemented in the programming language C. The sizes of ROIs were assumed to coincide with entire modules, and hits were generated at random locations, with $p$ determining relative frequencies. Thus benchmark results refer to the task of converting the amount of data that was contained in one ROI/module. Fig. 1.8(b) shows that absolute execution times for this type of pre-processing are in the same order of magnitude of those for proper feature extraction algorithms [36].

Pre-processing must therefore not be underestimated as a trivial adoption of shared conventions at an arbitrary late time during system design; it may require the provision of special hardware (perhaps similar in technology and design to what architecture A proposes up to the level of feature extraction — refer to section 1.3.6) or the provision of extra computing power in downstream general-purpose processors (up to 100% above “raw” estimates).

The results in Fig. 1.8(b) show that the formats with zero suppression are relatively superior in terms of introducing smaller delays during pre-processing for the targeted range of low occupancies. In a more general context, they indicate
that general-purpose processors (like the Sun workstation’s SPARC CPU) are ill prepared to deal with some formats, and that the task might better be given to more specialized hardware.
Chapter 2

Aspects of a specific 1996 technological implementation

The Soviet Union makes the finest microcomputers!
They are the biggest in the whole world!
Anonymous

Fig. 2.1 on the next page shows the components that served in an early implementation of a LVL2 trigger following architecture B (refer to section 1.3.7), when collaborating institutes brought together their equipment for common beam and lab tests at CERN in 1995/96.

These tests were meant as part of an exercise where adequate system performance had to be demonstrated by a combined program of physics simulation, prototyping, and modeling, prior to the construction of the final system.

Optimally, physics simulation defines the operational requirements of the system. Beam tests address integration problems with real detector prototypes. Lab tests allow to extract performance parameters from small-scale setups, which resemble only thin vertical slices of the LVL2 trigger system (for cost economy, each building block appears only once or in low counts), but already exhibit core aspects of system behavior. Modeling serves to extrapolate these results to the full size needed for the experiment, in order to yield decision support information. Results obtained from test lab measurements can enter the modeling effort both for calibration and validation purposes (refer to chapter 4).

The quoted setup used a combination of technologies from the vicinities of DSP320C4x Digital Signal Processors (DSPs) [72] from Texas Instruments and C104 Asynchronous Packet Switches [68] from Inmos/SGS Thomson (refer to section 4.3.2). At the time, these processors were considered as attractive choices for fulfilling the demands on parallel computing in the ATLAS LVL2 trigger, because of their on-chip communication ports, which promised a “glue-less” and straightforward mechanism for remote communication.

Several participating groups had past experience with both HEP experiments
Figure 2.1: proposed setup for lab tests at CERN in 1996
and transputers. When the T9000’s dim market prospects finally became evident also to its most ardent supporters, attention shifted to the then promising DSP320C4x generation of DSPs (and later to SHARC DSPs), because it featured similar advantages; these were delivered by up to 6 communication links per chip, in an otherwise more conventional architectural package. When the T9000 transputer was already gone for good as a device mainly for computing, its related C104 Asynchronous Packet Switch still survived, because of the lack of a similar device that could connect up large numbers of DSPs.

The similarity between the 2 technologies’ link types was thus put to test, and led to a particular interface design, where individual adapter cards used as many as 4 VME slots. This was due to a simplistic over-design which allocated 2 rather expensive microprocessors per interface line, and kept them exclusively busy with copying messages across (expensive) dual-ported memory in between, for accomplishing the required protocol conversion. This design ignored that a single chip (the C101) was commercially available for the same purpose as well (with the arguable exception of its lack of support for virtual channels, and thus lack of traffic shaping).

The reliability and performance of the overall system suffered, because of the DSP320C4x’s communication ports begin to fail when the cable impedance grows too large, which in practical terms for our system translated into maximum cable lengths of \( \sim 40 \text{ – } 50 \text{ cm} \) (no surprise there [72]). Yet the system setup made long cables unavoidable; it required such connections between 2 separate VME crates, or opposing ends of the same crate. (The C104-based switching network and its interfaces were housed in a separate crate from the beginning.)

This was a constant liability and led to an abundance of generally irreproducible error conditions, sometimes fugitive and resulting in transmission errors, but also frequently requiring re-initializations of the entire system. Typically most runs produced errors. Results from measurements taken on the system were unstable when they appeared, and were often dominated by the odd effects of patches for earlier problems.

In the following, I therefore took some liberty as to the scope of coverage of this by now out-phased system, and devote 2 sections to sub-tasks that contributed to its bottom-up growth and were successfully accomplished in their own rights.

2.1 An implementation of a TRT feature extraction algorithm for DSP320C4x Digital Signal Processors

A prototype of the Transition Radiation Tracker’s end-cap (end-cap TRT) [22] was used in the ATLAS beam tests during September 1995. It consisted of 5 sectors of \( 16 \times 16 \) straws each. Sectors were aligned in 5 left detector roofs
Figure 2.2: a sample event from the end-cap TRT prototype along the beam axis. Fig. 2.2 illustrates a typical sample event obtained from this setup. The different grey values correspond to pulse heights, which were measured on 2 different levels (low and high threshold hits).

Data for events that passed LVL1 arrived at the LVL2 trigger from 3 readout modules via 3 HIPPI lines. Each HIPPI line carried non-zero-suppressed straw information from 2 roofs and 3 consecutive bunch crossings. (Bits relating to the sixth, non-existent, roof were always set to one by definition.)

Fig. 2.3 on page 37 shows the protocol that was employed on top of HIPPI's own protocol for transferring sequences of 32-bit words in HIPPI packets and bursts [2]. The contents of the protocol’s header and trailer fields were irrelevant to the functioning of the feature extraction code, although the 2 counters in the first word added some convenience for identifying events in the data stream during debugging.

The readout modules had originally been designed to connect to sectors in adjacent left and right roofs. Because of a typical test situation, where every second sector was mounted in a left roof, but connected in place of a right one, different readout sequences followed for left and “right” roofs. The readout scheme in Fig. 2.4 on page 37 shows the ensuing 180° rotation; it makes use of the subdivision of roofs into channels (numbers 0 to 31, appearing before slashes) and daughter-boards (numbers 0 to 7, appearing after slashes) for uniquely identifying straws. Increasing channel numbers gave consecutive word offsets; daughter-board numbers and time slices determined bits within particular words. The protocol grouped information from a particular straw and 3 time slices (bunch crossings) into groups of 3 consecutive bits. The assignment between a group of
bits and individual straws was by increasing daughter-board numbers; i.e., the 8 most significant bits of each word remained unused. Under this scheme single-bit values from 8 straws were stored in single words; bits representing larger values were distributed over several words, as was the case for drift times (possible values 0 - 7; 3 words required).

In the test beam setup an ROI corresponded in size to the entire coverage of detector surface (5 roofs). The algorithm received RoID packets [78] from 3 separate ROIs in its first part (via a local switch in the LVL2 system); it extracted raw data blocks from these packets (starting from word 10) and merged them into blocks of 993 words or 3972 bytes. The raw data for the feature extraction algorithm thus consisted of 3 appended blocks of the format that is shown in Fig. 2.3 on the next page.

Because of the lack of other options, data pre-processing was performed as part of the feature extraction algorithm itself, although it is understood that this is not a likely architectural choice for the final trigger implementation, where more upstream devices will perhaps deal with the task. In the case of our implementation pre-processing amounted to data reformatting and zero suppression. Only information on hit straws was retained in an array of offsets into lookup tables. A discrimination of whether a straw was hit with high or low threshold was kept in the most significant bit at each offset. The lookup tables contained pre-calculated parameters, chosen such that the algorithm was able to carry out its principal calculations at maximum speed. Their semantics were as follows:

**offsets** contained a matrix of $16 \times 16$ words. It was used for conversion from the $\eta \times \phi$ coordinate system into a one-dimensional coordinate system that used offsets into lookup tables **lut0** to **lut6**.

**lut0** to **lut6** each contained $1536 \times 3$ numbers of bins (possible values 1 to 69 for the current parametrization) that were grown whenever a particular lookup table entry was addressed. Each entry consisted of up to 3 bin numbers, with zeroes identifying no bin at all.

**corr_lut0** to **corr_lut6** each contained 69 floating point values serving as bin height increments.

**inter_corr** contained 7 floating point values for normalization of bin heights (see below).

The actual feature extraction used a track finding algorithm based on a histogramming technique without clustering. Lines of 7 different slopes ($-15^\circ$ to $15^\circ$ in increments of $5^\circ$ relative to the beam axis) were drawn to intersect with hit straws in geometric space. Separate histograms for different slopes and for high and low threshold hits were used to accumulate the number of times such lines intersected the $\phi$-axis at particular offsets. The offset and slope of the best fitting
<table>
<thead>
<tr>
<th>HIPPI word</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2 counters for trigger number in burst</td>
</tr>
<tr>
<td>1 - 2</td>
<td>remaining header (Bx block count, run info, derandomizer full, derandomizer filling, TDM phase)</td>
</tr>
<tr>
<td>3</td>
<td>for 32 channels in left roof</td>
</tr>
<tr>
<td></td>
<td>high threshold bits</td>
</tr>
<tr>
<td></td>
<td>low threshold bits</td>
</tr>
<tr>
<td></td>
<td>drift times (LSBs)</td>
</tr>
<tr>
<td></td>
<td>drift times (MSBs)</td>
</tr>
<tr>
<td>163 - 165</td>
<td>trailer for left roof (parity check, trigger check, phd check)</td>
</tr>
<tr>
<td>166</td>
<td>for 32 channels in right roof</td>
</tr>
<tr>
<td></td>
<td>high threshold bits</td>
</tr>
<tr>
<td></td>
<td>low threshold bits</td>
</tr>
<tr>
<td></td>
<td>drift times (LSBs)</td>
</tr>
<tr>
<td></td>
<td>drift times (MSBs)</td>
</tr>
<tr>
<td>326 - 328</td>
<td>trailer for right roof (parity check, trigger check, phd check)</td>
</tr>
<tr>
<td>329 - 330</td>
<td>extra trailer</td>
</tr>
</tbody>
</table>

**Figure 2.3:** high-level protocol employed over HIPPI lines

**Figure 2.4:** readout sequence for straws in left and “right” roofs
track was determined from the location of the highest bin. Only low threshold information was used to establish this maximum.

The algorithm differed from other implementations mainly in the following regards:

- Unlike the TRT feature extraction algorithm in [36] it did not make any use of drift time information. The latter adds and subtracts drift times to/from straws’ values in order to get more precise coordinates. (Both signs have to be applied, because information about drift directions is lacking.)

- It accounted for the fact that lines of different slopes and offsets can hit only different maximum numbers of discretely spaced straws, by growing bins by pre-adjusted values (around 1), rather than by 1, and by normalizing bin heights from different histograms/slopes before final comparison. The first made sense also because of the DSP320C4x series DSPs’ ability to carry out arithmetic instructions in single CPU cycles, regardless of whether the argument is an integer or floating point number.

- As a facility to account for finite straw diameters, it allowed for up to 3 lines per slope to intersect the same straw, thus it grew up to 3 histogram bins per hit straw. This feature seems unattractive in terms of execution speed, because it adds another nested loop into the very core of the algorithm, although the algorithm takes care to execute only the required number of loop cycles. (Furthermore, the lookup tables lut0 to lut6 become significantly larger.)

The track finding algorithm returned the 5 features listed in Fig. 2.5 on the facing page. Floating point values were converted into integers, in order to avoid problems relating to different representations of floating point numbers on different CPUs. (DSP320C4x DSPs use a private floating point format by default.)

The last step of the algorithm served for packing the features into FEXD packets [78] (starting from word 8) and for sending these packets off to a processor in the global LVL2 farm (via the switch of the global LVL2 system).

The initial and trailing code sections dealing with RoID and FEXD records were written in C, the other parts were implemented in DSP320C4x assembly language for both performance reasons (the code was carefully optimized and laid out around delayed branches by moving instructions into branch delay slots, which serve a similar purpose as more recent RISC architectures’ predictive and speculative execution schemes) and historical reasons (early development was done on an installation without a C compiler). It is anticipated that these parts should be completely rewritten in C, in case that any significant modifications to the algorithm become necessary in the future.

As an offline testing procedure the native code was run with 74 input events from the ATLAS test beam of the type shown in Fig. 2.2 on page 35. A special
Figure 2.5: features extracted from the end-cap TRT prototype

program converted these events from records in a binary file into C source code. This was necessary, because the DSP’s C run-time library provides no direct support for accessing file systems. The results obtained from this simple test environment agreed well with our expectations relative to test beam data. In particular, the extracted slopes corresponded to the beam slopes, and occupancies were $\sim 4\%$.

2.2 A VMEbus-based server for DSP320C4x Digital Signal Processors

This section describes a server for down-loading programs to members of Texas Instruments’ DSP320C4x [72] family of Digital Signal Processor (DSPs). The software consists of 3 parts: The actual server program sunvme, suitable for execution on a Sun workstation that is acting as host, a stdio run-time library that provides ANSI C-like I/O functionality to programs executing on client DSPs, and a set of protocols for client-server communication over a VMEbus connection.

The server provides an alternative run-time environment to Texas Instrument’s source code debugger [71]. The decision for its implementation was taken, because our host was frequently crashing for some time, after the necessary modifications for installing the debugger’s drivers had been applied to the SunOS 4.1.3 kernel. These problems have been overcome in the meantime, so that — after gaining some experience with the 2 tools — we can now record specific advantages (and disadvantages) with both of them:

- With its commercial support the debugger is clearly the more mature software product and typically the better choice for dealing with programming
errors. It is popular among users because of its graphical user interface (GUI).

- On the other hand, only the server allows client programs to accept command line arguments and use a subset of ANSI C’s `stdio` library for printing messages to terminal windows or log files. This form of feedback is often of great advantage over having to inspect the contents of CPU registers or memory locations with the debugger. It proved valuable in particular for dealing with communication problems: combined status reports from several processors to a single terminal window usually allowed a better (if only coarser) status overview than what otherwise had to be established by clicking through multiple debugger windows.

Subsets of the `stdio` library are also available with more elaborate (and costly) commercial development environments (e.g. Parallel C [1], SPOX), but among the known solutions it is this server that delivers the same basic functionality with minimum penalty. (In particular, it does not require the DSPs to run real-time operating system kernels.)

The server’s batch-like mode of operation helped in automating downloading procedures. We felt that — especially as setups grew larger in terms of software running on several processors — there existed a balance and even a competitive advantage in usability over the debugger’s GUI.

Last but not least the server’s optional debugging mode, which caused output of detailed diagnostic information, (and the fact that its source code was available) helped us to overcome problems involving the somewhat shaky hardware setup and move past dead points on several occasions, even when the debugger was still used as the primary development environment.

The debugger requires a number of parameter settings before it can operate reliably. Some of these parameters (such as values for the Local and Global Memory Control Registers) potentially require different settings for each target site and thus frequent editing or copying of configurations files. The server does a better job than the debugger in assisting the user and providing a maintainable environment, because it can automatically provide some of these values and perform plausibility checks on others.

### 2.2.1 DSP320C4x Digital Signal Processors

The principal members of the DSP 320C4x family of Digital Signal Processors are the C40 and C44 DSPs [72]. Both implement 32-bit architectures with CPUs running at up to 50 MHz (25 MIPS), 128 word instruction caches, support for fast (single cycle) floating point arithmetics, and some parallel instructions. The DSPs features on-chip JTAG interfaces for debugging and control.
Their most characteristic feature are on-chip communication ports for rapid inter-processor communication. Depending on the type of DSP, 6 or 4 (C40 or C44) parallel ports can operate bidirectionally at speeds of up to 20 MByte/s in parallel, without attention required from the CPU. The ports physical and logical widths are 8 and 32 bits respectively. FIFOs for buffering up to 8 words are provided at each sending and receiving end. Ports are numbered from 0; after reset those with lower numbers (up to 2) are assigned output, the others input as their default directions. Port directions can in principle be arbitrarily reassigned at any later time, although the design of some boards imposes relevant restrictions (see below).

Every CPU instruction is encoded in a single 32-bit word. It is perhaps out of the need to maximize processors’ address spaces under this constraint that memory addresses where chosen to refer to individual words rather than bytes: thus Texas Instrument’s C compiler returns 1 (one word) for both sizeof(char) and sizeof(long).

The most direct effect of this uncommon (word-based) memory addressing scheme is an up to fourfold increase in memory consumption for character strings. If such “unpacked” character strings are transferred via communication links, only a fourth of the maximum achievable bandwidth is revealed to the naive user, who does not expect an infringement of such common programming standards. Depending on the exact nature of a program, the word-based addressing scheme can also lead to more subtle changes to semantics when code is ported from other (byte-based) platforms.

The DSP’s entire logical address space of 4 Gwords is split; its 2 equally large parts are connected to the CPU via physically distinct external address and data buses, known as local and global bus respectively. This potentially increases system performance by avoiding bus access conflicts. Different delays for software wait states and other characteristics can be programmed for 2 individual sub-parts each via the Global and Local Memory Interface Control Registers. The chip holds 2048 words of on-chip RAM (see Fig. 2.7 on page 43), which is mapped onto the local bus along with control registers governing communication ports, DMA channels, and timers.

Depending on the setting of 3 external pins, the chip can boot either from a set of predefined memory addresses or from one of its communication ports. If the second option is applied, a DSP listens to all of its communication ports in a round-robin fashion and selects the one where it first spots incoming traffic. Down-loaded software has to arrive on the selected port in a data stream that accords to the protocol shown in Fig. 2.6 on the next page [72]. It defines one or several blocks of executable code or data along with target memory addresses and values for several control registers.

The construction of values for the Global and Local Memory Interface Control Registers are discussed below with the different types of supported VMEbus boards. As for the other control registers, the server sets the values for both IVTP
<table>
<thead>
<tr>
<th>offset</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>value for Global Memory Interface Control Register</td>
</tr>
<tr>
<td>1</td>
<td>value for Local Memory Interface Control Register</td>
</tr>
<tr>
<td>2</td>
<td>specification of individual blocks</td>
</tr>
<tr>
<td></td>
<td>block size (number of words)</td>
</tr>
<tr>
<td></td>
<td>physical target memory address</td>
</tr>
<tr>
<td></td>
<td>block data (individual words)</td>
</tr>
<tr>
<td>0 (note that this value is distinct from any previous block size)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>value for Interrupt Vector Table Pointer Register (IVTP)</td>
</tr>
<tr>
<td></td>
<td>value for Trap Vector Table Pointer Register (TVTP)</td>
</tr>
<tr>
<td></td>
<td>memory address for IACK instruction</td>
</tr>
</tbody>
</table>

Figure 2.6: protocol for down-loading software via communication ports

and TVTP to 0x002ff800; this amounts to the definition of a single interrupt vector table at the start of the DSP’s on-chip RAM. An IACK instruction is used to flag the end of the boot sequence to the connected hardware; it requires an off-chip memory address that accepts dummy read requests, for which the server provides the address 0x00300000. While this is compatible with the range of currently available hardware (see Fig. 2.7 and Fig. 2.8), it lacks full generality.

### 2.2.2 VMEbus boards with DSPs

Texas Instruments has defined a TIM-40 standard, which specifies an interface between VMEbus-conformant carrier boards and daughter-cards (TIM-40 modules); it is on the daughter-cards where the DSPs are actually mounted. Several options are commercially available.

The DBV42 TIM-40 carrier board [57] provides support for up to 2 TIM-40 modules in sites labeled A and B. The 2 modules can exchange data both via dedicated communication ports (A4 and B0, interconnected via an on-board line) or via shared memory. One port per board (A2 and B5) is routed through the Link Interface Adapter circuitry (LIA) to the VMEbus; the remaining 4 ports are brought out to connectors on the board’s front panel. They are available for bidirectional off-board interconnections, subject to the restriction that physical connections must only be established between ports with different default directions.

The DBV42 is offered in different memory configurations (see Fig. 2.8 on
<table>
<thead>
<tr>
<th>size in words</th>
<th>address range</th>
<th>wait states</th>
</tr>
</thead>
<tbody>
<tr>
<td>internal RAM block 0</td>
<td>1K 0x002ff800-0x002ffff</td>
<td></td>
</tr>
<tr>
<td>internal RAM block 1</td>
<td>1K 0x002ffc00-0x002fffff</td>
<td></td>
</tr>
</tbody>
</table>

**MDC40S1 on-module (single width TIM-40 module)**

| external SRAM bank 0 | 32K 0x00000000-0x00030fff | 0 |
| external SRAM bank 1 | 32K 0x00030000-0x00030ffff | 0 |
| external SRAM bank 2 † | 32K 0x80000000-0x80007fff | 0 |

**MDC40S2 on-module (single width TIM-40 module)**

| external SRAM bank 0 | 128K 0x00000000-0x0031fffff | 0 |
| external SRAM bank 1 | 128K 0x00320000-0x0033fffff | 0 |
| external SRAM bank 2 † | 128K 0x80000000-0x8001fffff | 0 |

**MDC40HB1 on-module (double width TIM-40 module)**

| external SRAM bank 0 | 128K 0x00000000-0x0031fffff | 0 |
| external DRAM bank 1 | 1M 0x7fe00000-0x7fefffff | 1 |
| external DRAM bank 2 | 1M 0x7ff00000-0x7fffffff | 1 |
| external DRAM bank 3 † | 1M 0x80000000-0x800ffffff | 1 |
| external DRAM bank 4 † | 1M 0x80100000-0x801ffffff | 1 |

**MDC40HB2 on-module (double width TIM-40 module)**

| external SRAM bank 0 | 128K 0x00000000-0x0031fffff | 0 |
| external DRAM bank 1 | 4M 0x7fc00000-0x7fffffff | 1 |
| external DRAM bank 3 † | 4M 0x80000000-0x803fffff | 1 |

**MDC40HB3 on-module (double width TIM-40 module)**

| external SRAM bank 0 | 128K 0x00000000-0x0031fffff | 0 |
| external DRAM bank 1 | 4M 0x7f800000-0x7fbfffff | 1 |
| external DRAM bank 2 | 4M 0x7fc00000-0x7fffffff | 1 |
| external DRAM bank 3 † | 4M 0x80000000-0x803fffff | 1 |
| external DRAM bank 4 † | 4M 0x80400000-0x807fffff | 1 |

† depends on setting of hardware link

Figure 2.7: RAM options for on-chip and on-module memory
### DBV42 on-board (carrier board for 2 TIM-40 modules)

<table>
<thead>
<tr>
<th>size in words</th>
<th>address range</th>
<th>wait states</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared SRAM</td>
<td>128K</td>
<td>0x840000000-0x84020000</td>
</tr>
<tr>
<td>shared DRAM</td>
<td>1M, 4M, 16M</td>
<td>0x880000000-0x88fffffff</td>
</tr>
</tbody>
</table>

### DBV44 on-board (carrier board for 4 TIM-40 modules)

<table>
<thead>
<tr>
<th>size in words</th>
<th>address range</th>
</tr>
</thead>
<tbody>
<tr>
<td>external SRAM</td>
<td>128K</td>
</tr>
<tr>
<td>external SRAM</td>
<td>128K, 512K</td>
</tr>
</tbody>
</table>

† also shared with VMEbus
†† after shared bus has been granted
††† depends on setting of hardware link
†††† configurations with 20 ns or 70 ns RAMs available

Figure 2.8: RAM options for on-board memory

page 44). Its entire on-board RAM is shared between the 2 sites. Part of it is also visible to the host on the VMEbus and can be used alternatively for downloading software. The board’s LIA and other control registers (including one for its reset) are mapped into the A24:D32 VMEbus address space, starting from address 0x00400000. (Other address modifiers are supported, but not used by this server.) The base address is provided from an on-board PEROM, which can be reprogrammed for configurations with more than one DBV42. The board can act both as a master and slave on the VMEbus (depending on the setting of a hardware link).

The DBV44 TIM-40 carrier board [54] provides support for up to 4 TIM-40 modules in sites labeled A to D. This board has on-board communication lines between each module and 2 of its neighbors via ports A1 and D3, A4 and C0, B1 and C4, as well as B4 and D0 respectively. One port per board (A2, B5, C5, and D2) is routed through the Link Interface Adapter circuitry (LIA) to the VMEbus; the remaining 3 are brought out to connectors on the board’s front panel. They are available for bidirectional off-board interconnections, also subject to the restriction that physical connections must only be established between ports with different default directions. The DBV44 does not have any on-board RAM. Its LIA and other control registers (including one for resetting the board) are mapped into the A24:D32 VMEbus address space, starting from an address that is configurable via hardware links. The board acts as a slave on the VMEbus.
The MDC40S [55] and MDC40HB [56] are daughter-cards of single and double width respectively. Both adhere to the TIM-40 standard, carry one DSP, and come in different RAM configurations (see Fig. 2.7 on page 43).

The RHUL board [12] is a specially designed VMEbus board with a single directly mounted DSP. Its VMEbus slave interface allows access to a 2K block of dual-ported SRAM for data exchange between host and DSP. The SRAM is mapped into the DSP’s address range such that its base address matches one of the addresses from which the DSP can alternatively fetch its boot information. Unlike what is the case for the DBV42 and DBV44, this board does not route one of its DSPs’ communication ports to the VMEbus, nor can it be reset by software. In return for disobeying the TIM-40 standard, the board offers some other features: all 6 communication ports are brought out to the front panel (although they can only be used in their default-directions), port 4 has an extended (off-chip) input FIFO for 512 bytes, and an optional broadcast mode mirrors incoming traffic from input port 5 to output port 2.

The values for the Global and Local Memory Interface Control Registers, which the server needs to provide for each type of target (see Fig. 2.6 on page 42), depend on the amount and types of external memory connected to a DSP in a given embedded environment. Under the TIM-40 standard the DSP’s local bus is used for accessing on-module memory; its global bus is used for accessing a mixture of on-module and on-board memories, depending on the exact configuration (see Fig. 2.7 and Fig. 2.8). Hence, the proper value for the Global Memory Interface Control Register depends on both the types of carrier boards and TIM-40 modules, whereas the proper value for the Local Memory Interface Control Register can be determined from the type of TIM-40 module alone.

Fig. 2.9 on the next page [54, 55, 56, 57] shows values for individual fields that enter the calculation of the value for the Global Memory Interface Control Register in the case of the DBV42 or DBV44. For the RHUL board (which does not obey the TIM-40 standard) the value is constant at 0x32e4b990 (assuming 70 ns RAMs) [12].

The values for the Local Memory Interface Control Register are 0x3deba050 [55], 0x3dec2050 [55], and 0x1e704000 for MDC40S1, MDC40S2, and MDC40HB TIM-40 modules respectively. In the case of the RHUL board the server uses 0x3def7880 [12].

2.2.3 SBus-to-VMEbus interface

Fig. 2.10 on the following page shows the principal communication paths available between hosts (on the left side of the drawing) and DSPs (on its right side). Both Texas Instrument’s debugger and the server make assumptions about preferred paths among the available options. In this section we briefly discuss these options and give a rationale for the selections that we took.

For the sake of this discussion it is convenient to regard hosts as falling into
<table>
<thead>
<tr>
<th>field</th>
<th>carrier boards</th>
<th>TIM-40 modules</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DBV42</td>
<td>DBV44</td>
</tr>
<tr>
<td>STRB0_SWW</td>
<td>—</td>
<td>0x01</td>
</tr>
<tr>
<td>STRB1_SWW</td>
<td>0x03</td>
<td>0x02</td>
</tr>
<tr>
<td>STRB0_WTCNT</td>
<td>—</td>
<td>0x00</td>
</tr>
<tr>
<td>STRB1_WTCNT</td>
<td>0x01</td>
<td>0x07</td>
</tr>
<tr>
<td>STRB0_PAGESIZE</td>
<td>—</td>
<td>0x1e</td>
</tr>
<tr>
<td>STRB1_PAGESIZE</td>
<td>0x09</td>
<td>0x19</td>
</tr>
<tr>
<td>STRB_ACTIVE</td>
<td>—</td>
<td>0x0f/0x10</td>
</tr>
<tr>
<td>STRB_SWITCH</td>
<td>0x01</td>
<td>0x01</td>
</tr>
</tbody>
</table>

Figure 2.9: field values for Global Memory Interface Control Register

![Diagram](image)

Figure 2.10: clients and servers in the test lab setup

46
2 different categories, as depending on their amount of support for remote client access and on the exact type of physical connections to the VMEbus boards.

UNIX hosts (such as Sun workstations) most seamlessly allow client access from remote sites on the Internet, because of their support of the TCP/IP protocol stack and of multitasking. (Remote usage of Texas Instrument’s debugger additionally requires X Windows support, because of the debugger’s GUI.) A remote access capability is highly desirable, because it potentially allows a large and geographically scattered user community to share access to a common set of DSPs (e.g. from outside CERN).

Hence we defined UNIX workstations as our preferred host system. As for other operating systems that we use in our laboratory, adding support for LynxOS would be easiest, because it requires neither an SBS915 device nor associated drivers. Porting the same software to a PC would be most difficult in comparison, because of the general lack of availability of documentation on the DSPs’ JTAG implementation. Our particular choice of a workstation from Sun was determined by the availability of the SBS915 SBus-to-VMEbus adapter board (see below). Our choice of the SunOS over Solaris 2.x operating system was determined by the versions of the drivers /dev/ptvme* (see below) that we had access to.

Two distinct types plus a mixed solution exist for establishing physical communication paths:

One is via a line of JTAG connections. JTAG is a standard for in-circuit emulation and analysis, which is supported on chip level by the DSPs and on board level by JTAG connectors on the front panels of carrier boards. Starting from a host that is equipped with a suitable JTAG interface card (it’s typically a PC running DOS or OS/2) clients can be daisy-chained via JTAG.

The second type of communication path is via the VMEbus: a portion of VMEbus address space is shared between the host and its clients.

In the case of hosts whose address space is entirely embedded in the VMEbus address space, access to VMEbus memory is straightforward. This is typically the case for VMEbus-based processor boards that run real-time operating systems, such as LynxOS or OS/9. In the case of Sun workstations (and similarly other systems with activated memory protection) an SBS915 SBus-to-VMEbus adapter board [66] is required. This interface defines several memory regions (windows) whose base addresses remain constant in SBus address space, but are arbitrarily movable in VMEbus address space under software control. This functionality is available on 2 levels of software abstraction: one uses a modified kernel and a set of reentrant drivers /dev/ptvme* (preferred option); the other one is by directly manipulating the board’s control registers through the drivers /dev/sbus* that come with Sun’s original kernel.

In order to be useful for our purposes, this memory area must allow for bidirectional data exchange between the server and clients and for the passing of reset signals to clients. In the case of the DBV42 and DBV44, which map one communication port into VMEbus-accessible memory through their Link Interface

47
Adapters (LIAs), this type of connection allows the server to channel data into a communication port, which is convenient for down-loading software. In particular, it is more so than down-loading to and booting from memory, because the sizes of VMEbus-accessible memories and their amount of overlap with addresses from which the DSP can boot varies between carrier boards.

In the case of the RHUL board, which does not make any communication port accessible on the VMEbus, functionality that is similar to the LIAs can be added by making use of the Link VME Interface (LVI) [28] board. It provides connections for 4 outgoing and incoming ports each on its front panel and maps them into a configurable region of A32:D32 VMEbus address space. Thus a LVI board (and pairs of unidirectional communication lines to front panel connectors on RHUL boards) can serve as functional replacements for up to 4 LIAs.

The mixed solution relies on a DBV42 or DBV44 board and its JTAG emulation facility. Connections between one of these boards and the host are via the VMEbus; connections between the board’s DSPs and more downstream clients are via JTAG lines.

Texas Instrument’s debugger requires this type of connection. However, we did not choose to support it, because it were exactly the related software drivers /dev/ptume* and /dev/dbu* for the SBS915, DBV42, and DBV44 boards that caused severe problems to our host system. At the time the host did only work reliably when the drivers were absent, for otherwise all invalid VMEbus memory accesses would crash the entire (multiuser) system. (An addressing error in a down-loaded client program was enough to trigger the effect.) Even though this is now no longer the case, the drivers are still non-robust against intermediary power losses on VMEbus boards, as they for instance occur when crates are briefly shut down for maintenance purposes. The resulting effect is still highly inconvenient, because the very nature of our test lab environment necessitates frequent modifications to the hardware setup and thus power switches.

The drivers /dev/sbus*, /dev/ptume*, and /dev/dbu* essentially form a hierarchy in the sense that later ones require and operate on top of the previous. Texas Instrument’s debugger demands all 3 sets of drivers, while our server is satisfied with only the first one (this requires no modifications to the original operating system kernel at all) or a combination of the first 2 (following standard driver installation procedures).

2.2.4 COFF files

Texas Instrument’s C compiler, assembler, and linker [73, 74] store images of executable code in files that adhere to the COFF (Common Object File Format) specification. This is a well established standard (it’s also used by the host’s own operating system, for example) that allows for portability and upgradability by splitting the file contents into mandatory and optional headers (both of fixed sizes), a table of section headers with further pointers to section contents, and a
symbol table (of variable size).

The down-loaded code is extracted from sections with the following names: 
.text contains executable code and floating point constants, .const contains pre-initialized data, .stack and .sysmem contain the run-time stack and heap respectively, and .bss holds data that needs to be pre-initialized with values stored in the .cinit section. These values are provided as sequences of words, where every entry consists of at least 3 words: the first defines the length of the value in number of words, the second contains the physical target address for initialization; actual values start from the third word. 0 in place of the first word denotes the end of a sequence.

The server performs initializations and down-loads all sections in turn. It makes sure that the .text section is down-loaded first, because this is where executions of client programs begin. In addition, it also patches code at one address. This is required for passing command line arguments into the function main without having to change Texas Instrument’s run-time library. The library contains a call to main (the underscore identifies the assembly language version of a C function) at an offset of 0x16 from the symbol _c_int00. The patch modifies this call and replaces it with a call to main:patched: the effect function behaves as an extra piece of code in the run-time library, which gets control before the execution of the C program on the client starts. Its purpose is to read command line arguments from a communication port (together with other information: see Fig. 2.12 on page 51) and to prepare the parameter stack before yielding control back to main.

The server assumes that the small memory model (not: big) and standard parameter passing conventions (not: register-argument passing) are used for compiling client programs. During development we employed versions 4.50 and 4.60 of Texas Instrument’s C compiler, and we carried out numerous successful tests. It is well understood that this kind of low level patching is extremely sensitive to any changes in future versions of the C compiler’s run-time library. Therefore, the server performs consistency checks and uses a highly conservative evaluation strategy before committing to any modifications.

2.2.5 Protocol for down-loading executables

The data stream described in Fig. 2.6 precedes additional data that correspond to a protocol shown in Fig. 2.12 on page 51. The initial data stream is mandatory and required for running any software on DSPs. The latter one has been defined to conveniently suit the server’s specific demands. Its purpose is to pass additional information from the server to its clients and aid in the following regards.

- The server uses communication ports for down-loading software and for communication with clients. While DSPs on DBV42 and DBV44 carrier
boards can make use of bidirectional communication lines via the Link Interface Adapter (LIA), in the case of RHUL there is no way for the client to know which port the server is listening on, because different (unidirectional) ports have to be used for communication between a host and a client. Accordingly, the server uses the protocol to prescribe specific port numbers to client programs that run on RHUL boards.

- Source and destination addresses are typically expressed as port numbers in communication software that has been written for DSPs. This differs significantly from the approach that is taken in higher-level communication libraries, such as MPI (refer to section 3.1.2.1) and others, where sources and targets are identified by process numbers (ranks), in order to achieve higher flexibility and adaptability to changing topologies. Any implementation of a similar library on DSPs requires information on the given network topology, for mapping between ranks and port numbers.

The server passes the required information to its clients as part of the protocol, after extracting it from 2 sources: it implicitly knows about (unchanging) on-board connections on DBV/4/2 and DBV/4/4 carrier boards, and it reads information about other connections from a topology description file.

This text files consist of lines following the syntax shown in Fig. 2.11. All characters following a hash sign on the same line are ignored as comments. The parameters board no, have been included for allowing compatibility with future versions of the software, which may support more than one instance of each board. (Their values are presently ignored.)

- The server uses the protocol to pass command line arguments to client programs, which are executing under the server’s control.

### 2.2.6 Protocol for receiving output from client programs

Each client program must include the version of the standard header file `stdio.h` from the server’s directory, and it must have been linked against the correspond-
<table>
<thead>
<tr>
<th>offset</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>protocol version (0 for this version)</td>
</tr>
<tr>
<td>1</td>
<td>non-zero, if server is running in debugging mode</td>
</tr>
</tbody>
</table>
| 2      | port selection for communication from client to server  
   |   0 - it’s up to client to choose a port number  
   |   1 - server wants client to remain silent during operation  
   |   2 - server recommends specific port number (at offset 3) to client |
| 3      | recommended port number (if any) |
| 4      | rank of down-loaded process within process group |
| 5      | size of process group |
| 6      | for all processes in the process group, ordered by increasing ranks  
   |   site on which process is running  
   |   1 - site A on DBV42  
   |   2 - site B on DBV42  
   |   3 - site A on DBV44  
   |   4 - site B on DBV44  
   |   5 - site C on DBV44  
   |   6 - site D on DBV44  
   |   7 - site on RHUL board  
   |   board number (always 0 in this version)  
   |   for all 6 ports, ordered by increasing port numbers  
   |   site which output from this port goes to  
   |   0 - port cannot act as output port  
   |   others - see above  
   |   port’s priority among output ports connecting to same site  
   |   board number (always 0 in this version)  
   |   site which input to this port arrives from  
   |   0 - port cannot act as input port  
   |   others - see above  
   |   board number (always 0 in this version)  
   |   port’s priority among input ports connecting from same site  
   |   for all C command line arguments  
   |   string length of i-th command line argument argv[i] |
   |   i-th command line argument argv[i] without terminating ‘\0’ |

Figure 2.12: protocol for communication from server to clients
typedef enum {
    SITE_NONE = 0,
    SITE_DBV42_A, SITE_DBV42_B,
    SITE_DBV44_A, SITE_DBV44_B, SITE_DBV44_C, SITE_DBV44_D,
    SITE_RHUL
} SITE;

int fprintf (FILE *stream, char *format,...);
int printf (char *format,...);
int putchar (char c);
int vfprintf (FILE *stream, char *format, va_list ap);
int vprintf (char *format, va_list ap);

int sunvme_rank (void);
volatile COMPORT_REG *sunvme_in_port_per_rank (int rank);
volatile COMPORT_REG *sunvme_out_port_per_rank (int rank);
int sunvme_site (SITE site, int board_no);
int sunvme_size (void);

Figure 2.13: prototypes defined in stdio.h

The selection of implemented functions was tailored according to present needs, which resulted in support for only a small subset of the entire stdio library at this stage. If a more complete subset will be required in the future, corresponding source code is readily available from the GNU C compiler\(^1\) and similar source code distributions.

\(^1\)ftp://prep.ai.mit.edu/pub/gnu/gcc-2.7.2.tar.gz

The selection of implemented functions was tailored according to present needs, which resulted in support for only a small subset of the entire stdio library at this stage. If a more complete subset will be required in the future, corresponding source code is readily available from the GNU C compiler\(^1\) and similar source code distributions.

\(^1\)ftp://prep.ai.mit.edu/pub/gnu/gcc-2.7.2.tar.gz

fprintf(), printf(), vfprintf(), vprintf() are compatible to their ANSI C counterparts with the following restrictions: stream must only be set to stdout or stderr; i.e., general file output is not supported. The conversion specification in format strings must specify the type of conversion only as one of the letters c, d, f, s and x [49]. Field width, precision, and other options from the ANSI C standard [49] are not yet supported.

putchar() is compatible to its ANSI C counterpart.

sunvme_site() gives the size of a process group, i.e. the number of client programs that have been launched into execution from a single invocation of the server.
sunvme\_rank() gives a unique process identifier by returning a number between 0 and sunvme\_size().

sunvme\_site() informs a process about the site it is running on. A non-zero value is returned, if the actual site matches the specification given by the 2 actual parameters.

sunvme\_in\_port\_per\_rank() and sunvme\_out\_port\_per\_rank() return pointers to the Communication Port Control Registers of the port that accepts input from (sends output to) another client process with a given rank. The return value is established based on information that has been passed from the server to all clients (see Fig. 2.12 on page 51). If there is no (know) physical connection between the 2 relevant sites, a pointer to the first input (output) port is returned as default value.

The protocol that is used for communication from clients to the server has deliberately been kept simple. It consists of the raw sequence of ASCII characters that client programs print to stdout or stderr. (The server effectively merges both streams, before their output actually appears to the user.) 0 serves as simple escape character: if it is followed by a non-zero value, it indicates termination of a client program, in which case the second following byte is interpreted as the program’s exit code.

After down-loading software the server continues to run either until all its client programs terminate with exit codes 0 (indicating no error), or until at least a single one terminates with an exit code different from 0 (indicating errors). The server buffers output from clients such that individual lines contain output from only one source. If more than one client is running at the same time, the server also prefixes output lines with site names, in order to enhance readability.

The motivation behind this simple protocol was that connections between servers and clients were often found to be unreliable. For instance, a long lasting problem with LVI boards caused corrupt data that arrived from clients with systematic errors that included duplicated words in the data stream. (This did not affect the traffic that was bound for clients, probably because it was sent by the host in single burst without pauses.) With the current simple protocol (without any packing or other abstraction) output from the client could typically still arrive in the host’s text window in a readable form.

Choosing this simple protocol of course caused cuts that affected other functionalities: it restricted client programs to using only single output streams. Thus file output was not supported, neither was any form of input streams. A slightly richer protocol for communication from clients to the server (see Fig. 2.12 on page 51) could nevertheless easily include provisions for multiple streams and for client access to the host’s file system. Candidates for this type of protocol could resemble the conventions that characterize the communication between a
Figure 2.14: summary of command line options supported by sunvme

UNIX kernel and device drivers for character-oriented devices. (In particular, representation conversions into character strings should remain as clients’ tasks.)

2.2.7 Command line options

Fig. 2.14 gives an overview of the command line options that the server comprehends. Their semantics are detailed below.

args causes the server to pass all remaining command line arguments (also those starting with hyphens) to the client without interpretation. The client program can access them as arguments in accordance to the ANSI C standard, by defining a program entry point that matches the following prototype.

\[
\text{main} \ (\text{int} \ \text{argc}, \ \text{char} * \text{argv}[\]);
\]

dbv42 instructs the server to down-load an executable onto a TIM-40 module on a DBV42 carrier board. site (possible values a and b) identifies the target module, tim identifies the type of TIM-40 module (possible values hb1, hb2, hb3, s1, and s2) and filename refers to the COFF file which stores the executable. If filename is without an extension, .out is appended by default.

dbv42_base1 tells the server about the DBV42’s base address in VEEMbus A24:D32 address space. If the option is not specified, the server assumes 0x00400000 as default value.
dbv42_reset explicitly resets the DBV42 board. This also happens implicitly, when the server attempts to down-load an executable to any of the board’s sites.

dbv44 instructs the server to down-load an executable onto a DSP that is mounted on a TIM-40 module on a DBV44 carrier board. site (possible values a to d) identifies the target site, tim identifies the type of TIM-40 module (possible values hb1, hb2, hb3, s1, and s2), filename refers to the COFF file which stores the executable. If the filename is without an extension, .out is appended by default.

dbv44_base tells the server about the DBV44’s base address in VMEbus A24:D32 address space. If the option is not specified, the server uses the default value 0x00000000. (This is an arbitrarily choice; the DBV44 does not define any particular default base address.)

dbv44_reset explicitly resets the DBV44 board. This also happens implicitly when an executable is down-loaded to any of its sites.

debug causes output of verbose debugging information from the server and its clients.

little_endian prevents the server from byte order swapping while down-loading the contents of COFF files. Texas Instrument’s Sun-based development tools record COFF files using big-endian byte order (definition: address of a datum is address of its MSB). This seems natural for the host’s SPARC-based processor architecture, but is alien to the DSPs’ little-endian target architecture (definition: address of a datum is the address of its LSB). The server provides the necessary byte swapping conversion by default. Texas Instrument’s PC-based development environment version produces little endian COFF files instead, which causes the necessity for an explicit toggle.

lvi_base tells the server about the LVI’s base address in VMEbus address space. If the option is not specified, 0x0f000000 is assumed.

rhul instructs the server to down-load an executable onto a DSP on a RHUL board. i and j specify the output and input ports on the LVI board (possible values 0 to 7), k and l specify the output and input boards on the RHUL board (possible values 0 to 5). Proper operation requires that physical connections have been established between the front panel connectors identified by i and l, as well as j and k. filename refers to the COFF file which holds the executable. If filename is without an extension, .out is appended by default.
**sbs915_leds** turns the LEDs on the SBS915 board’s front panel on, according to the binary representation of $n$. If the parameter is not specified, all LEDs are turned off. This option serves for debugging purposes; it is only available, if `-sbus` is chosen at the same time.

**sbs915_reset** generates a SYSRESET signal on the VMEbus. This bold intervention is currently the only way to reset RHUL boards under software control. It must be applied with caution, since it will cause resets to all boards connected to the VMEbus. This option is only available, if `-sbus` is chosen at the same time.

**sbus** instructs the server to access the VMEbus via `/dev/sbus/*`, rather than via the `/dev/ptvme/*` family of drivers. (Use of this option should be restricted to circumventing errors that occur in the default mode.)

**topology** tells the server which topology description file it should use. If `filename` is not given, `topology` is assumed by default. If the option is absent, the forwarded topology information lists only on-board connections.

**upper** instructs the server to print the output from every second client in upper case letters. This helps to distinguish between output from different clients.
Chapter 3

Technology tracking and communication benchmarks

Computers are my forte!
LIME (in Brazil)

We defined benchmarking of parallel computer systems as a preparatory step towards choosing the right architectural and technological implementation for the ATLAS LVL2 and LVL3 triggers. As such, it complements other activities that continue to go on in parallel, such as the build-up of lab-scale prototypes (refer to chapter 2) and discrete event simulation (refer to chapter 4). By taking these efforts’ combined results (after suitable extrapolation both in scale and technology) into account, planners will then approach the ultimate measure of comparison, consisting of some combination of cost, complexity, commercial availability, etc.

The mix of strategies and their interplay is specifically meant to help in attacking the fundamental complication that any suitable final solution for high-frequency triggering in ATLAS has to face during its design. The size of the experiment is too big to allow for the construction of large-scale physical prototypes. Yet at the same time rapid and ongoing technological advancements make it infeasible to predate decision making and construction to such an extent that basic design modifications can be scheduled as undetermined results from future on-hands experience will dictate.

In addition, technology assessment and tracking is also important, because the very trigger structure will critically depend on performance characteristics of available components, which cannot be defined in abstract.

In order to advance our studies, we defined a comparatively naive set of programs to form a benchmarking suite. It allowed us to assess the communication performance of commercially available parallel computers and highlight their behavior in situations that are specifically important to our real-time application.

The motivating idea behind these benchmarks is to evoke communication semantics that are similar to those envisaged in candidate designs for the final
ATLAS LVL2 trigger. Therefore their parameter space was defined such as to englobe various potential ATLAS traffic patterns. They remain at a sufficiently abstract level, such that implementations can be brought forward with only modest effort, and that results for a variety of potential options can be achieved and meaningfully compared before future buying decisions have to be made.

An important aim was furthermore to provide a focus for benchmark activities in the context given by our intended application, and thus to help avoiding unnecessary duplication of effort and proliferation of benchmarks.

In addition, the same benchmark suite has been picked up for other purposes as well, such as pre-purchase evaluations of parallel computers by a large computing center.

### 3.1 A taxonomy of multiprocessor architectures

Machines like the Cray T3E, IBM SP-2, Meiko CS-2, SGI Origin, and others follow a trend that has become predominant in general-purpose large-scale computer and supercomputer design in recent years, and that has almost eliminated the previous generation of monolithic vector supercomputers (or at least restricted their use to areas which can be characterized as largely insensitive to financial considerations). We use the term supercomputer as a convenient abbreviation for machines that are used in the high-end scientific and engineering marketplaces.

Today’s multiple instruction and multiple data stream computers (MIMD — this terminology was first introduced in [23]) typically offer better integration of recent mainstream technology, scalability, overall price-to-performance ratio, and reuse of existing software by applying a decomposition into a moderate number (tens to hundreds) of commodity microprocessors and (still) particular interconnects as their main construction principle.

Because of the fact that industry can provide high-performance microprocessors off-the-shelf, these designs benefit from the mainstream market’s large production quantities, for decreasing risks and cost of new products and for yielding competitive prices.

Although for marketing reasons microprocessors are less aggressively optimized for raw speed than the special CPUs that served in earlier supercomputers (for instance they use CMOS over gallium arsenide implementations), the competitive nature of the workstation market and the fact that microprocessors are manufactured by hundreds of thousands rather than by the dozen, progressively led to an important overall advantage for designs that incorporated mainstream advancements and kept their counts of privately built components low. Recession and shrinking resource funds in the U.S. at the end of the cold war also had an important catalytic and consolidating effect.

\footnote{http://www.csc.fi}
Figure 3.1: growth in microprocessor performance over the past decade

Fig. 3.1 (reproduced from [39]) elaborates on the same point by showing the strong amount of growth of microprocessor performance during the past decade. The graph uses measurements that are relative to the SPECint92 benchmark suite; gains in floating-point performance were even stronger during the decade. While up to the mid 1980s advancements were primarily technology driven, the following accelerated growth rates originated from more advanced architectural ideas.

Fig. 3.2 presents individual ideas in a spectrum between evolutionary (typical scenario: preserve binary compatibility or require only re-compilation of existing code) and revolutionary (typical scenario: new programming models, languages and algorithm design needed) in effect.

On the software side these MIMDs are better guarded against what has caused limited success and short life for many preceding designs (the Connection Machine [40] was a typical example that attracted much attention at its time), because individual processing nodes in MIMD multiprocessors often appear as re-packaged workstations. (Often each node runs a separate copy of a modified version of the UNIX operating system.)

Unlike vector supercomputers, whose design reflected typical engineering problems, these MIMDs can draw from a large existing base of workstation applications and customers. They can support large user bases and run mixed workloads of both conventional and parallel applications; this conveniently allows for smooth transitions during “down-sizing” of services from mainframes.

Their design helps to tolerate the invariant fact that only a limited amount of parallelism is available per application, therefore Amdahl’s law imposes a fixed
Figure 3.2: Evolutionary versus revolutionary trends in computer design.
Although there are some recent standardization efforts for system area communication, existing solutions are still largely in the proprietary domain of individual vendors. This situation is reasonable, because of the late onset of related technological developments in comparison to uniprocessors. It is also tolerable, because the semantics of communication can effectively be hidden behind ordinary CPU instructions (shared memory programming model) or inside standard software libraries (message passing programming model). These abstractions however come at a price that is analogous to run-time overhead that is caused by interpreted code.

The following compilation lists 3 related (de-facto) standardization efforts that operate on different levels.

- The Intel Pentium Pro is an example of a recent microprocessor with on-chip arbitration logic, which allows several (up to 4) CPUs to cohabitate the same system bus and share a coherent view of memory through their caches. This has reduced the yardstick for implementations of small-scale parallel systems in a particular technology. A feedback effect might become visible in the near future, when CPU producers may put more than one microprocessor on a single chip (super-chip), thus shifting the new trend back into the mainstream. This type of standardization comes as a “free” extension, but makes narrow demands on the type of microprocessors (uniform), media (proprietary system bus), and scale (small).

- The SCI [47] standard (refer also to section 3.3.4) proposes protocols and interfaces for allowing heterogeneous nodes to contribute to a pool of commonly shared memory. This level of standardization tolerates requirements from different vendors and suggests one particular set of conventions for interconnecting these platforms.

- The Virtual Interface (VI) architecture specification is an initiative to define standard hardware and software interfaces for system area networks (SANs). A system area network is a specialized network optimized for the reliability and performance requirements of clusters of servers and workstations that differs in performance and reliability from features offered by LANs and WANs. The aim of the VI architecture is to spur innovation in SAN technology and to make the LAN, WAN, and SAN differences transparent to the application, without depending on a particular medium, microprocessor, operating system, or programming model. A draft technical specification is currently under preparation with backing from Compaq, Intel, Microsoft, and companies identified with technologies such as ATM, SCI, Ethernet, and others. In summary, this level of standardization aims at encompassing variations in the types of technologies of both nodes and networks.

---

2http://www.intel.com/procs/servers/isv/vi/v2
Fig. 3.3 (reproduced from [67] with modifications) suggests a taxonomy for categorizing the memory organization of MIMD multiprocessors, which is itself similar to Flynn’s [23] more general scheme in utilizing 2 dimensions with binary scales. The abbreviations that occupy its square fields are explained in the following sections.

The first dimension (vertical axis) deals with the physical organization of memory. A distinction is made whether memory resides as a single entity at a central place or whether it is physically distributed in separate fragments along with the processors. The second one (horizontal axis) deals with whether memory is logically regarded as a single pool (shared memory programming model) or as a set of scattered and disjoint resources (message passing programming model).

The following discussion adopts the logical discrimination and is therefore structured by reference to 2 different programming models.

### 3.1.1 Shared memory programming model

The shared memory programming model allows processors to rapidly exchange information by accessing global memory locations via normal CPU load and store instructions. It comes to an important advantage of this model, that these instructions do not involve the function call overhead that must be tolerated with the competing message passing programming model.

The solution also offers conceptual simplicity by allowing systems to be interpreted in terms that are already familiar from uniprocessors. (The software abstractions offered by standard UNIX IPC facilities and related extensions, such as POSIX threads [51], are in fact fully sufficient for establishing controlled access
to shared memory resources.) The (complicated) structure of the network, which mediates between processors and memory, is largely hidden from applications by an elaborate memory sub-system.

For some implementations relative simplicity is an illusion, which is effectively maintained by the sub-system for the benefit of application programs, and that is met by a measured degree of sophistication in its hardware design. A broad categorization of memory sub-systems can be drawn as follows.

3.1.1.1 Uniform memory access

The simplest shared memory multiprocessors use designs that put several processors along with DRAM memories on the same system bus (bus-based multiprocessors). As was already illustrated with reference to the Intel Pentium Pro, some contemporary microprocessors include on-chip logic for enabling cooperative memory sharing of this type. Since the access paths from all processors to memories are of equal lengths (one hop across the system bus), these architectures are termed UMA (for Uniform Memory Access).

All contemporary microprocessors use a hierarchy of caches that mirror a small part of main memory for faster data exchange with the CPU (SRAM technology, multi-ported, sit on the same side of the system bus and close to a CPU). In multiprocessors caches no longer are arranged only “vertically”, in the memory hierarchy at different offsets between CPUs and DRAMs, but also “horizontally”, along the medium that interconnects CPUs. A non-trivial complication arises, because the memory sub-system must guarantee that the set of caches presents a single coherent view of memory contents to their processors at all times, while allowing for multiple copies of cached data to exist.

As a result, cache coherency protocols must be devised for updating memory locations all at once in potentially many caches that hold local copies. Unlike in uniprocessors (without consideration of I/O), caches must no longer only take note of memory updates from “upstream”, as effected by write-through from the CPU that sits next to them, but also from “downstream”. The latter occurs when listening to traffic on the system bus reveals that cached memory locations are changed by another processor (snooping) or by active notification (directory-based schemes).

UMAs suffer from limited scalability in terms of number of connected nodes, because the single system bus increasingly becomes a congested bottleneck, as more and more processors compete for arbitration and memory accesses in the case of cache misses. Adding local cache memories to every processor can avoid some bus traffic, but the bus still tends to remain as hot-spot, as node counts increase. Electrical design considerations also put limits on the total sizes of backplane buses. Today’s UMAs are built with configurations of between 2 and dozens of microprocessors. (Some implementations, such as the Cray CS6400 Enterprise Server, somewhat alleviate the restriction on scalability by using several
parallel buses instead of one.)

Fig. 3.4 (reproduced from [25]) suggests that shared memories in UMAs continue an earlier trend in varying trade-offs between memory sizes, distances, speeds, and cost. This trend is also manifest in the conventional hierarchy of caches, and its extension therefore maps well onto the memory access patterns (locality) of existing applications.

3.1.1.2 Non-uniform memory access

An attempt to achieve higher scalability over bus-based multiprocessors (UMAs) quickly identifies their single system bus as the main hot-spot, because every cache-miss (frequent) or cache update (very frequent), which is triggered by any out of an increasing number of processors, intensifies contention for the exchange of information (memory contents) or meta-information (cache coherency protocol tokens) via a single shared medium.

An obvious solution is to replicate the sparse resource and distribute the main memory among the processing nodes to provide scalable memory bandwidth. A small portion of the entire memory is then local to each processor (and connected to it via a private system bus). The access performance of this local memory portion is similar to what is found in a uniprocessor, while the latency to access data in remote portions can be by orders or magnitude larger, hence the term Non-Uniform Memory Access (NUMA).
Existing systems are often hybrids that use a two-tier memory hierarchy to achieve scalability. Local groups of processors (nodes) have access to portions of shared memories with UMA characteristics. The remainder must be accessed remotely across an interconnection network (NUMA).

The entire machine's hardware essentially resembles a network of complete nodes, where each node's memory sub-system detects remote from local addresses, and routes related bus transactions via the network to the node which owns the requested memory portion. The used networks typically utilize topologies with many point-to-point connections instead of a linear, bus-like structure. The grouping of multiple processors per node helps to amortize the cost of each node's network interface.

One question that arises in the design of such large-scale single-address-space machines is whether to allow caching of shared writable data. The advantage, of course, is that caching allows higher performance to be achieved by reducing memory latency; its disadvantage is the now enlarged problem of cache coherency. Snoopy cache coherency protocols are no longer adequate, because of the (potential) lack of a broadcast medium.

SCI-based systems (refer to section 3.3.4) and the Stanford DASH [53] prototype opt for cache-coherent NUMA memories (CC-NUMA). Their directory-based (instead of snoopy) cache coherency protocols afford higher scalability through point-to-point notification services. The protocols’ relative complexity is however considerable, and for instance evident from the fact that Dolphin's SCI-to-PCI adapters, which we had available for testing (refer to section 3.3.4), lacked related implementations and functionality. Rather, remote memory locations simply remained uncached, and each read from a location, whose virtual address identified it as remote, led to a new transaction, which can of course yield highly undesirable results in naive code.

In the absence of caches applications have to take the underlying heterogeneous nature of global shared memory into account for achieving satisfactory performance, by for instance avoiding short-cycled loops on remote flags. Thus the performance of this type of NUMA machines can be severely limited, if data is incorrectly placed in memory locations away from the nodes that use it most, because data cannot migrate properly.

In particular, many existing large-scale machines do not support cache coherency. Alternatively, they rely on assistance from development tools, such as compilers and linkers, for matching the coherency demands of particular applications with explicitly generated code for its maintenance, or by placing shared variables with different access semantics (private, read only, or read/write) in different types of global memory (cached or uncached).

As is traditionally the case with machines from its manufacturer, the Cray T3E, which is available in configurations of up to 1024 nodes, operates without the provision of cache coherency.

Memory Channel [25, 26] opts for a design choice that is similar in effect
(refer to section 3.3.3). Because of its asymmetric assignment of read and write permissions to communicating peers (see Fig. 3.16 on page 103) the problem does not arise in its usual form, but propagation delays of transactions through the network must now be taken into account, and receivers are therefore still obliged to actively detect and ensure memory coherency (distinguish updated values from old values).

For CC-NUMA, data from remote nodes can only reside locally in a node’s caches, never in its main memory (DRAM). This feature restricts the total amount of data that each node can replicate from other nodes at any given instant and leads to a pressure to build very large (and expensive) off-chip caches when designing CC-NUMAs [67].

Cache Only Memory Architectures (COMAs) take a radically different approach by converting each node’s local memory into a (slow) associative cache of similar size. The notion of a permanent relationship between ranges of physical addresses and particular nodes disappears, and data are automatically copied and migrated to physical memory in the node(s), where they are actually used. The Data Diffusion Machine (DDM) has put this idea into practice, but its applicability suffered from the inherent requirement for completely non-standard memory sub-system hardware and associated costs.

Distributed Virtual Shared Memory (DVSM) systems in turn aim at providing a global shared memory abstraction on distributed hardware solely by manipulations on the level of system software, over what CPUs’ standard MMUs already provide. Like in a COMA, a DVSM node’s main memory is used as a local cache of the machine’s global virtual address space. References to remote addresses that are not already cached are revealed to the operating system by page faults. The ensuing interrupt handling code fetches the requested page across the network (rather than from disk), in a process that is quite similar to virtual memory handling in uniprocessors. (The relationship between physical and virtual memory is itself an embodiment of caching.) DVSM systems are in principle very attractive, because they are coupled to very modest hardware requirements, such that they can operate on anything from tightly coupled MPPs to networks of workstations. But their implementation suffers from comparatively poor performance; it also provokes false sharing and contention, because their unit of sharing is at the granularity of physical pages (considerable coarser than cache lines). Secondary effects from handling page faults in software, such as the pollution of caches with operating system data, can further decrease application performance [67].

### 3.1.2 Message passing programming model

The message passing programming model requires that both senders and receivers actively and explicitly participate in communication by calling send and receive subroutines from system communication libraries. This is in contrast to the shared memory programming model, where communication is one-sided and
implicit.

In its most literal form, each message passing call involves a (re)synchronization between 2 peers; execution stalls in preparation of a rendezvous, until both processes issue communication requests by making the relevant subroutine calls.

The message passing programming model does not provide the abstraction of a single and globally shared address space, but equips parallel processes with communication channels for data exchange between their private address spaces. All data migration and duplication is under the sole responsibility of applications, such as all issues relating to memory coherency as well.

The paradigm is mainly attractive because of its wide portability and scalability. It is easily compatible with both single address space multiprocessors (UMA and NUMA), with MPPs, NOWs, and with combinations of these elements. Its scalability is for instance evident from the fact that the socket API to the Internet’s TCP/IP protocol stack adopts a very similar practice. In contrast, the competing shared memory programming model is not well supported on MPPs and NOWs.

There is also a natural correspondence between its messages and methods that form interfaces between private scopes (address spaces) of classes under the object-oriented programming paradigm. CORBA and other facilities for remote procedure calls (RPCs) constitute manifestations of the message passing programming model for object-oriented software construction in distributed computing environments.

Active Messages use optimized remote procedure calls that specialize in extracting data at receivers’ ends and feeding them into ongoing computations in proper context-sensitive ways, as determined by compilers. They have the potential of boldly reducing software overheads that are normally associated with orthodox versions of the message passing programming model, but under present practice rely on the adoption of particular tools, such as Split-C over C.

In our terminology MPPs (Massively Parallel Processors) are similar to clusters of workstations in first approximation. Both typically run separate copies of their operating systems and feature private address spaces per node. The IBM SP-2 is an example of an MPP system.

Workstations on any network segment can be configured to form a network of workstations (NOW), as language semantics trivially suggests. They use low-cost commodity networks, such as Ethernet, FDDI, Fiber-Channel, or ATM, as their main discriminating feature. This is in contrast to the use of proprietary networks in MPPs. NOW’s network interfaces are loosely coupled to I/O-buses (e.g. PCI). This is again in contrast to the tightly coupled network interfaces, which often serve in MPPs, and which are directly connected to the memory buses of participating nodes.

Whereas shared memory systems implement means to reduce latency (coherent caches), the message passing programming model offers means to potentially tolerate latency through non-blocking communication calls (latency hiding).
Non-blocking send and receive functions allow for the overlap of message transmission with computation, or the overlap of multiple message transmissions with one another. They come in pairs: posting functions begin the requested operation; test-for-completion functions allow the application program to discover when a requested operation has come to an end. The semantics of posting functions are streamlined such as to permit quick returns.

While latency reduction through the provision of caches is a manifest and predetermined feature of a given hardware, latency hiding can be improved by stronger efforts during software development. Because of the greater flexibility of latency hiding, recent shared memory architectures are also beginning to provide support for message-like block transfer mechanisms (e.g. packet mode of SCI), a feature normally associated only with the message passing paradigm.

Although further development of SANs will lead to lower absolute latencies, the relative importance of latency hiding will perhaps increase, because of the even faster growth pace in microprocessor performance. (Their relative past improvement rates as compared to DRAM memories present evidence for a similar scenario.)

Because of the existence of multiple private address spaces, the message passing programming model implies that senders (and receivers) have no control over the use of remote addresses by their peers. In particular, remote addresses have no meaning (and no discernible representation) in machine-level instructions (no remote memory access — NORMA). The offered level of abstraction, with subroutines that implement send and receive primitives, utilizes extra parameters, in addition to those passing local addresses, for identifying remote nodes.

Function call overheads, multiple data movements, and crossings of the protection boundary between user level and operating system level constitute serious performance drawbacks that have to be tolerated when the message passing programming model is in use and frequent small messages prevail (fine-grained parallelism). The hardware overhead of shared memory architectures (in particular CC-NUMA) therefore quasi makes its reappearance as high software overhead: complexity is redistributed but preserved.

In retrospect, the 2 programming models exhibit the following cardinal differences.

- The shared memory programming model presents processes running on different nodes with a uniform address space, which they may share. This potentially enables zero-copy (or one-copy) implementations of communication libraries, where only pointers to data rather than data themselves are exchanged. This evidently leads to a reduction from $O(n)$ to $O(1)$ in the complexity that is typically associated with data exchanges between processes.

- It delegates communication entirely to the user level and thus removes the need to frequently cross the protection boundary between user processes and
The execution of a single system call in UNIX typically involves an overhead that is in the order of magnitude of several microseconds, and thus already close to minimal latencies that modern interconnects aim to achieve.

- The shared memory programming model is one-sided. Apart from protection issues, it does not require software participation on 2 nodes in order for communication to happen. This is much in contrast to the strict rendezvous semantics in the message passing programming model.

- It reduces software overheads, as caused by network protocols and network drivers, to such an extent that single user-level store (or load) instructions, which are executed by the CPU (e.g., MEMORY CHANNEL or transparent mode of SCI), or a DMA engine (e.g., packet mode of SCI) can initiate data transport across the network. Software security checks only need to be applied once when cluster-wide shared resources are requested. (This requires kernel involvement and is therefore undesirable on a per-message basis.) Once “circuits” are set up, the standard virtual address translation hardware takes over and ensures integrity via the memory mapping and protection mechanisms that are inherent in contemporary operating systems, such as flavors of UNIX and Windows NT.

- The message passing programming model makes communication explicit in source code; it thus helps with debugging and software maintainance by exposing hidden semantics and by restricting race conditions and similar problems that are typically difficult to reproduce, diagnose, and track down, to fewer places of possible occurrence.

Partly in contrast to this argument it has been observed that the shared memory programming model allows programmers to concentrate on algorithm development rather than requiring their attentions for communication issues. This is because they can view the network as a natural extension of conventional memory. Thus the convention leads to better reuse of algorithms, when the transition to multiprocessors is made.

- In order to run parallel programs where processes access shared resources it is imperative to provide some form of synchronization primitives in addition to the basic data exchange primitives as well. Under the message passing programming model both responsibilities are naturally combined within the semantics of communication primitives. This allows for stable application-level code. (The communication library is written and debugged only once per many application life cycles.)

Under the shared memory programming model on the other hand, communication has an attractive resemblance to pure memory accesses, but
synchronization typically still requires subroutine calls. (E.g. calls for manipulating mutexes via the POSIX threads library [51].) If the programmer chooses a conservative strategy to protect against race conditions, the number of related calls will reach approximately the same count that was before negatively attributed to the message passing programming model.

Optimizations can be achieved, in particular, if the shared memory programming model is used as a compilation target for languages such as Ada [7], which allow for abstractions of parallelism on the level of dedicated syntactic constructs. Ada includes support for both paradigms via rendezvous — message passing programming model — and global variables — shared memory programming model —, although these syntactic forms do not necessarily mandate corresponding underlying hardware models. Java also qualifies under this bracket.

Because of the omni-presence of pointers in C and C++ code, these languages require that the programmer keeps full responsibility over synchronizing access to shared data structures at all times (hindrance for implicit programming model).

- Message passing permits the use of a single uniform programming model in heterogeneous environments, where diverse communication technologies such as e.g. SCI, MEMORY CHANNEL, ATM, Gigabit Ethernet, shared memory (this is for instance convenient for prototyping parallel software on a single workstation which hosts many processes), Ethernet (for slow control), and others might be present, and where only some choices support operations on cluster-wide shared memory.

The metaphor of a chain which is only as strong as its weakest member usefully applies. Message passing can be implemented on top of shared memories with modest cost in performance, while the reverse typically requires hardware support. (One can regard MMUs and for instance also SCI interfaces as devices for achieving this reverse transition, where “messages” — swapped virtual pages in one case, and SCI transactions in the other — are transparently hidden from client applications; refer to section 3.3.4.)

3.1.2.1 MPI — Message Passing Interface

The Message Passing Interface (MPI) [61] is an application programming interface (API) that defines the syntax and semantics of a core of library routines for reliable inter-process communication by way of exchanging explicit messages, as is generally foreseen in the message passing programming model.

The recent past was characterized by a market situation where each vendor supported mainly a private API for offering message passing functionality. Although many of the general semantics approached similar concepts, their syntax
and precise semantics differed between various sets.

This amounted to a competitive disadvantage of the message passing programming model in comparison to the shared memory programming model. In the latter the hardware of the memory sub-system takes full responsibility over presenting a coherent view of memory, whereas with message passing a layer of intermediate software interfaces and protocols (middleware) is required to mediate between distributed memory resources.

Thus the message passing programming model potentially has to gain a lot from preferentially using a common software standard at this level. A portable, reliable, and widely used standard would free vendors’ resources from doing overlapping work and reduce the lag times for bringing new parallel systems to market.

Adherence to a well-balanced standard also adds flexibility to the development process (wider choice of tools) and eases software maintenance (availability of trained manpower). Portability permits that development and debugging outside (expensive) production environments and conveniently adds more vendors and systems to conceivable paths for software migration.

This optimistic view is reflected by the fact that a consortium of over 40 computer manufacturers, industrial users, and academic institutions collaborated in preparing the first version of MPI. They drew on experience from vendors’ private solutions and from the similar PVM (Parallel Virtual Machines) effort in the past. Rather than selecting one and adopting it as standard, the body sought to make use of the most attractive features from a number of existing libraries. Due to its mode of creation, MPI pragmatically opts for largely adopting existing practice.

The standard was in particular completed with a self-imposed deadline, and known deficiencies were left for coverage in an updated version MPI-2 [62]. This also avoided preemption of useful investigations by too early imposition of relating rules.

MPI attempts to achieve both conformity with many hardware platforms (portability) and low software overhead (efficient execution) by the following arrangement: the standard defines more than 150 functions on the outset. Since considerable care has been taken to streamline them into adherence of a common style, their functionality can alternatively be compressed into a much smaller set of core functions.

Minimal implementations can choose to emulate the redundant functions by mapping them to sequences of core operations. High-performance implementations can alternatively shortcut emulated code, if special hardware support in selected areas exists. An operation for broadcasting an identical message to several receivers serves as a typical example for redundant, yet often desirable, services.

MPI’s basic functions for point-to-point communication (send and receive primitives) take process ranks, communicators, tags, buffers, their sizes, and MPI data types as their arguments. They provide reliable and ordered (non-
overtaking) communication, and perform automatic data type conversion in heterogeneous network environments.

MPI uses integer process ranks for uniquely identifying all processes in a consistent manner, with sequential numbers starting from 0. A program can use several overlapping enumerations at the same time, by making use of MPI communicators.

Communicators are a flexible aid for the implementation of reusable libraries, whose local assumptions on absolute process ranks should be encapsulated from client applications. Apart from grouping processes, each communicator is also associated with a separate communication domain, where messages do not interfere with those from other domains (different “virtual” or physical media). Thus communicators introduce scoping mechanisms for communication and naming contexts for processes. Unique process ranks can be discovered by inquiring them from specific communicators. The default communicator MPI_COMM_WORLD encompasses the set of all processes.

Tags allow a second level of selectivity for messages at the receiving end: on the first level receivers can accept messages from any sender (possible use: server applications) or restrict their attention to a particular sender. With tags they can furthermore impose a selection that is similarly based on message types, rather than origin (possible use: high-priority message relay).

Conceptually, both communicators and tags are means for partitioning possible streams of communication into distinct classes of scope and priority. Wildcards can serve as place-holders for tags, but communicators must be discretely identified at all times.

MPI offers 2 kinds of communicators, i.e., intra- and intercommunicators. MPI_COMM_WORLD is an example of an intracommunicator. Neither kind can be created from scratch, but only after manipulating an existing communicators’ set of processes with set manipulation operations.

Intracommunicators allow bidirectional point-to-point message passing between all pairs of included processes, and collective operations on arbitrary subsets of these processes. The possible connections between processes in an intracommunicator form a complete graph. Intracommunicators can optionally be associated with process topologies; these may convey rules for mapping processes to distributed resources to the run-time system, and enhance the code’s readability, by introducing process orderings that are natural for given topologies.

Intercommunicators connect members of 2 disjoint groups of processes. (Intracommunicators can obviously be regarded as special intercommunicators where both groups happen to be identical.) The allowed connections between processes in an intercommunicator form a bipartite graph, where each process in a local group is connected to all processes in a remote group. Intercommunicators can only be used for point-to-point communications. The corresponding functions retain the same syntax as compared to the use of intracommunicators, but specified ranks of sources and destinations are treated as relative to remote groups in
MPI is more concerned than its vendor-specific ancestors with support for heterogeneous parallel systems. These may include different types of nodes at potentially remote physical sites. This support is revealed by the presence of arguments for conveying type information in many MPI calls. MPI uses this information to perform transparent representation conversion, when necessary (e.g., between nodes with little- and big-endian architectures).

As a library, rather than a compilation system, MPI depends on the programmer to provide suitable type information. Future versions of MPI might take advantage of polymorphism in object-oriented languages, in order to pass data type information implicitly. The library at present includes functions for constructing representations of complex data types at run-time. These include non-contiguous arrays, among others. By applying a single communication call to an argument describing multiple sections of a multi-dimensional array at once, fixed communication overheads can thus be amortized over the transmission of many array elements.

In addition to its use of type information, MPI provides explicit operations for packing and unpacking as well. This mostly adds backward compatibility to earlier message passing libraries, such as PVM. The associated disadvantage is that explicit packing and unpacking require additional memory-to-memory copies. This turns PVM into a three-copy system in the presence of non-contiguous data structures, whereas 2 copies suffice for MPI (refer to section 3.2.2).

MPI's basic communication primitives are offered symmetrically for a variety of modes. Choosing a particular mode has indirect effects on latency hiding, requirements for local buffer space, number of protocol steps between senders and receivers, and so forth. The creators of MPI argued that programmers should be free to choose between all modes; one choice of semantics is not necessarily the best for all applications, nor is it the best for all architectures. Narrowly enforcing the use of particular modes in all cases thus would lead to semantic gaps with undesirable effects on performance and proliferations from the standard. The provision of a multitude of send and receive primitives for each mode partly explains the large overall number of functions in MPI ($\gtrsim 150$).

The following discussion draws a distinction between major and minor modes, which always appear in pairs. Actual modes are revealed by the (optional) second and (ubiquitous) first upper-case letter that appears in the quoted function names after the common prefix. The following are major modes.

**blocking** (MPI_Bsend, MPI_Ssend, MPI_Rsend, MPI_Send) The return from a call to a blocking primitive indicates that all resources that were specified in the call are available for immediate reuse by the application. Any visible change in the state of the calling process as affected by a blocking call therefore occurs before the call returns. With respect to communication, this entails that messages were either moved to intermediate buffers, which are

their case.
non-blocking (MPI_IBsend, MPI_ISsend, MPI_IRsend, MPI_ISend) A return from a call to a non-blocking primitive leaves the application free to perform other tasks, but these must not yet include (re)use of the same resources. A non-blocking call may initiate changes in the state of the calling process that actually take place after the call returned. This complication competes against the specific advantage that non-blocking calls typically take only short times to complete. Non-blocking functions allow the potential overlap of communication and computation, or the overlap of multiple communications that go on in parallel (refer to section 3.1.2). They require 2 relating calls, which are typically arranged in brackets around inserted calculations. The semantics of a single blocking call is effectively split into 2 parts: the first call initiates communication but does not yet await its completion. This can for instance relieve CPUs from having to wait idly for the completion of a DMA operations. The second call deals with completion; versions that unconditionally await completion (MPI_Wait) or return immediately if the indicated communication is still pending (MPI_Done) are available.

Minor modes (see Fig. 3.5) effect mainly the buffering policy that MPI employs on the senders’ sides.

buffered (MPI_Bsend, MPI_IBsend) The communication library uses intermediate buffer space, as provided by the application program, for decoupling remote send and receive operations. While this allows both peers to proceed largely independently, it causes additional memory-to-memory movements.
synchronous (MPI_Ssend, MPI_ISsend) This mode enforces rendezvous semantics between senders and receivers. Therefore each communication coincides with a (re)synchronization of the participating processes at their points in codes where the corresponding function calls return. The completion of a send operation thus also indicates information about the state of the receiving process, and vice versa. Synchronous mode has the extra benefit of functioning without buffering and its cost.

ready (MPI_Rsend, MPI_IRsend) This mode serves for the frequent case that matching receive calls were posted before the corresponding send calls occurred. By narrowing the allowed semantics, it thus allows for the removal of handshaking protocol steps and results in improved performance.

standard (MPI_Send, MPI_ISend) Implementations are free to choose any previous minor mode as their default mode of operation. Usage of standard mode by the programmer implies that he does not make any assumptions that depend on a particular mode, and that the implementation is therefore free to choose the most accurate mode under the circumstances. The aim of this mechanism is to enable the most straightforward mode, if the correctness of the application program is not concerned by the choice. Depending on whether buffer space is tight, send operations in standard mode may behave like either their buffered (with buffers now owned by the communication library) or synchronous counterparts.

In addition to its point-to-point communication primitives, MPI also supports collective communication (see Fig. 3.6 on the next page). The term “collective” in this nomenclature indicates that all processes in a group must invoke the same function. During instances of collective communication messages are transmitted among all processes in a group, as specified by an intracommunicator which is passed as argument. If one process acts in a special role (such as the sender of a broadcast message), it is identified by an additional argument that passes its rank.

The syntax and semantics of the collective communication primitives are consistent with point-to-point communication primitives. Partly in order to keep the (already large) total number of functions in MPI within reasonable bounds, they are nevertheless more restrictive in several ways. Only blocking versions are supported, receivers must specify exact (rather than maximum) packet sizes, and tagging is not available. Their remaining semantics is analogous to the standard mode for point-to-point communication. Thus collective communication primitives need not have (but may have) the additional effect of synchronizing the participating processes. A primitive for performing barrier synchronization without passing any data provides an exception to this rule.

Global reduction operations in MPI (see Fig. 3.7 on the following page) resemble collective communication operations that additionally perform numerical
buffers

processes

MPI_BCAST

MPI_SCATTER

MPI_ALLGATHER

MPI_ALLTOALL

MPI_GATHER

Figure 3.6: global communication operations in MPI

Figure 3.7: global reduction operations in MPI
operations of the type $f(a_0, a_1 \ldots a_{n-1})$ “on the fly”. Available choices include standard operations such as $\text{sum, min, max}$, as well as user-defined functions. These operations gather remote input arguments from all members (intracom-communicator) and distribute results alternatively to one or to all participating processes.

The follow-up standard MPI-2 [62] enhances the described capabilities of MPI in mainly the following regards.

- dynamic process management for the creation and termination of processes during run-time: the number of processes in $\text{MPI_COMM_WORLD}$ is no longer fixed
- one-sided communication primitives (get or put instead of send and receive), which map efficiently onto capabilities of shared memory systems or interrupt-driven communication systems
- non-blocking collective communication and collective communication with intercommunicators
- language bindings for C++ and Fortran 90, in addition to C and Fortran 70 as well

### 3.2 ATLAS communication benchmarks

In order to perform performance measurements of several computer network technologies, we gave ourselves a set of standard programs, which served as our main software tool, and which are referred to as ATLAS communication benchmarks in the following.

A deliberate design decision in favor of a simple and portable set of programs was taken in order to facilitate the quick accumulation of measurements for many representative systems [11], including existing equipment and emerging prototype implementations, in a reliable and reproducible fashion and at acceptable cost.

The selection of algorithms in the benchmark suite was in particular determined by the desire to isolate semantic contents to such a degree that results from these programs can reveal a set of comparable technology performance numbers, such as overhead and asymptotic bandwidth, for each particular candidate technology.

From the results of these measurements, planners should be able to identify possible bottlenecks in the intended system implementations, and derive numbers corresponding to a crude and abstract trigger model with comparative ease.

We tried to avoid the often-met difficulties that typically arise from interpreting abstract performance numbers that bear similar names (e.g. latency), but stem from variously interested sources, by requiring an interpretation that was
strictly in relation to these well-defined algorithms. In particular, we focused on end-user-to-end-user performance numbers including all overheads. What was measured is therefore a combination of features that can be attributed to influences from hardware, system software, a low level interface (middleware), and application semantics.

We were also interested in using a metric that reflects system conditions that can be extrapolated to large numbers of nodes with long lifetimes of operation. In particular, the contributing software must also be acceptable for use in realistically large systems, which therefore excludes over-optimistic fine tuning for specific benchmarks.

A complete implementation\(^3\) of the benchmark suite is available in the programming language C. Although C is by no means a good specification language in a general sense, this implementation also served pragmatically and conveniently as a canonical reference version, against which exact parameter semantics can be quickly checked.

Ideally, the obtained performance numbers will substantially improve the input to current computer models and serve during their calibrations. These models can then in turn be used to explore the interrelation of technological and architectural choices in a “virtual” system at large. As a result from modeling, formerly abstract parameters will be transferred into application-specific metrics, whose interpretation can assist in the final comparison of candidate technologies.

Our particular design decisions translated into the following set of assumptions.

- Experience so far shows that bottlenecks in the ATLAS LVL2 trigger’s design will more likely originate from communication, rather than from an overall incapacity to perform local algorithms at sufficient rates. (A full scan of the TRT sub-detector, as for instance required during low-luminosity runs, is one example to the contrary.) Accordingly (and since we wished to decouple our progress from still ongoing developments in detailed definitions of algorithms and detector optimizations), the benchmarks’ designs reflect the pragmatic assumption that communication performance depends only on traffic patterns, packet sizes, and processor occupations, but is independent of their information contents (such as specific hit patterns in sub-detectors). Therefore the algorithms have no detector or physics contents, but merely form skeletons for communication.

- Any implication arising from the need to interface these algorithms to specific hardware or detectors is not addressed. Therefore all data streams are allowed to originate in memories.

- In order to encourage support and participation from many hardware vendors and to enhance the fairness of comparisons between results (by keeping

\(^3\)http://www.cern.ch/RD11/combench/combench.tar.Z
the number of factors that affect comparisons low), we provided application-
level code that can run on many platforms as-is. The design emphasizes
portability over fine-tuning at the application-level by isolating technologyspecific parts in a thin communication layer, which serves the main code
base via a well-defined low level interface (refer to section 3.2.1).

The decoupling of how programs formulate their communication semantics
from specifics of particular network technologies, by means of using a com-
mon interface, successfully facilitated the spread of an integral code base
and the gathering of results for many different hardware platforms.

In addition to exposing the desired key parameters, the benchmarks also pro-
vided a convenient and reliable test-bed for newly emerging technologies and
prototypes at several times. They constitute a single yardstick against which oper-
ational stability and amounts of software integration can be measured. During
system testing one can utilize a set of existing programs for populating an exam-
ple application layer. This allows to perform isolated and well controlled tests
of the underlying hardware and middleware with sufficiently mature programs,
whose behavior is well understood from previous evaluation on a range of other
platforms. Thus apart from their quantitative aspects, the ATLAS communication
benchmarks also allow for valuable proofs of concepts. Since their low
level interface essentially forms a subset of MPI, a large number of programs is
potentially available for inclusion in elaborate test procedures.

The chosen algorithms can be segregated into 2 large groups; this is reflected
by a numeric nomenclature with major and minor indices (before and after peri-
ods). Seven generic algorithms (1.x) measure basic communication parameters.
They are complemented by 5 application-specific benchmarks (2.x and 3.x), 2 of
which (3.x) are comparatively newer in origin and reflect more recent thinking
concerning the trigger’s organization. The specific subset of those 2 benchmarks
is referred to as DAQ61 benchmark suite in the following [9]. The application-
specific benchmarks in particular mimic traffic in different proposed architectures
of the trigger system.

In addition to the fixed semantic contents of individual benchmarks, their
emerging traffic can be further defined by a number of parameters, whose values
are passed in by the following set of command-line parameters:

```
com- [1-3].x -c c -d d -l l -p n -r n, -s n, -w n

ns, nw, nr and n quote the number of senders, workers, receivers (where
appropriate), and the total number of processes that partake in communication.
l specifies the size of messages in bytes (ignored by some benchmarks), c chooses
the repetition count, and d specifies computing loads in units of microseconds.
```

Rather than putting processes to sleep for the requested amounts of time
d (operating systems are often ill equipped to provide precise services at mi-
crosecond granularity), they enter waiting loops for adequate numbers of cycles.
“Waiting” inside these loops amounts to incrementing values of type `unsigned long`. The programs include code that dynamically calibrates these wait-loops at startup time, based on the relative performance of CPUs. This has been found to operate accurately (e.g. $1600 \pm 10 \mu s$).

All benchmarks programs were tried in series of measurements in order to explore a relevant section of the trigger’s design space. This procedure in particular emphasized the frequent passing of numerous small to medium sized (few kByte) messages.

The individual benchmarks obey the following semantics (see Fig. 3.8 on the next page).

one-way (1.1) The total number of processes involved in this benchmark is fixed at 2; they act in split roles as sender and responder respectively. The measurement consists of observing the time $t_s$ that is required for a full round-trip of a single message of $l$ bytes from and to the sender via the responder. Dividing this time into 2 gives an estimate of the latency between 2 user-level processes executing at remote ends of the communication network.

two-way (1.2) Again, 2 processes are involved, but this time the same code is executing twice (it corresponds to the sender’s in one-way (1.1)). The measurement consists in observing the average time $\bar{t}$ for sending and receiving (2 different) messages of $l$ bytes. Concurrent flow of outgoing and incoming traffic streams is supported by the use of non-blocking communication primitives. This benchmark can take advantage of potential bidirectional communication links for achieving higher bandwidths.

all-to-all (1.3) The total count of processes involved in this benchmark is an even number $n$. All processes are simultaneously senders and receivers, and peers are chosen at random. This benchmark helps to study network congestion and scaling properties. Given an accumulation of results for a series of node counts, a parameter $\alpha$ for describing the deviation from a system that scales ideally can be gained from the relationship $t(kn) = \alpha kt(n)$.

pairs (1.4) $n/2$ dedicated pairs of processors partake in unidirectional communication across a shared network. Unlike with all-to-all (1.3), there is no contention by multiple senders for fewer receivers, but only for sparse resources inside the medium. This benchmark helps to study the network scaling properties with little or no contention.

outfarming (1.5) and multicasting (1.6) These benchmarks measure the extent by which one sender can communicate efficiently with $n_r$ receivers.
Figure 3.8: ATLAS communication benchmarks
While outfarming (1.5) reflects a situation with different message contents, multicasting (1.6) explores a network’s hardware broadcast capability, where available.

**funnel (1.7)** The benchmark requires that \( n_s \) senders forward messages to one receiver, thus temporarily provoking extreme congestions. In ATLAS parlance this benchmark is important for assessing a technology’s event-building capacity (refer to section 1.3.3).

**push-farm with supervisor (2.1) and pull-farm with supervisor (2.2)** One process acts as supervisor which informs \( n_s \) senders about the shifting identity of one out of \( n_r \) receivers (round-robin scheduling). The measurement accumulates times over a number of steps, with initial communication starting at the supervisor; this is followed by messages passing from several senders to the appropriate receiver (immediately or on request); the mimicked event loop completes with a receiver informing the supervisor about a “decision”.

**pipeline (2.3)** This benchmark assumes an arrangement of \( n \) processes in a two-level, narrowing pipeline, which consists of stages with processes that act as senders, workers, and receivers respectively. All processes try to propagate messages through the pipeline as fast as possible, only flow-control mechanisms that are implicitly enforced by the network apply. In ATLAS parlance this mimics a system that fully exploits parallelism both at the levels of sub-detectors and ROIs, by “geographically” allocating corresponding numbers of permanent resources.

**active ROBs and push-farm (3.1)** \( n_s \) data sources send messages of \( l \) bytes each to a single receiver. The receiver collects all messages and then performs pseudo calculations for durations of \( d \) microseconds. When \( d \) is set to 0, this benchmarks behaves indistinguishably from funnel (1.7).

This one and the following benchmark are implemented such as to allow a moderate amount of overlap between computation and communication, inasmuch as they start receive operations for fragments belonging to the \((n+1)\)-th event, before they begin non-preemptive calculations on the \(n\)-th event. (The presence of a DMA coprocessor typically causes a large effect with regard to this potential.)

**passive ROBs and pull-farm (3.2)** Its semantics is similar to the previous benchmark, with the exception that data sources send messages (1024 bytes) only after receiving prior requests from receivers. Request messages have a fixed size of 64 bytes. Event computation is carried out during the “time of flight” of requests and fragment that relate already to the next event. Receivers send one set of requests, however they do not request fragments...
belonging to more than one future event in advance. This is partly because only few implementations of the low level interface (or underlying software) allow for multiple outstanding send and receive operations between 2 processes at each time.

Requests are not broadcast, but rather sent out as many point-to-point messages. This is because message contents differ per destination and because the communication layer defines only a blocking broadcast primitive (following MPI also in this regard: refer to section 3.2.1). The latter is undesirable with respect to overlapping computation and communication.

There is no explicit process (such as an ATLAS LVL2 supervisor) that deals explicitly with flow control; rather it is left as a task to the underlying communication layer. In active ROBs and push-farm (3.1) senders always attempt to send at full speeds. When receivers cannot keep up, the communication library performs intermediate buffering and/or stalls sends until the required resources reappear.

The present implementation does not try to enable configurations with multiple sets of senders and/or receivers by the provision of dedicated source code. Many related studies of network interference can be done with less effort by exploiting mechanisms for explicit scheduling as provided by the run-time environment that comes with particular communication libraries.

Fig. 3.9 shows example topologies that can be activated with MPI via the provision of different host files alone.

3.2.1 Low level interface

Fig. 3.10 on page 85 shows the protocol stack that enabled the ATLAS communication benchmarks. The various characteristics of individual benchmarks
form the horizontal aspect of the uppermost layer. The vertical scale includes a common low level interface as its most prominent ingredient.

This layer of the protocol stack implements support for the message passing programming model. It offers its functionality to client applications as a code library. In the terminology of the OSI reference model it provides the transport layer’s services, with reliable and ordered end-to-end process communication. The layer supports calls that have been carefully chosen such as to resemble a core of communication primitives from the industry-standard MPI [61] message passing library (refer to section 3.1.2.1).

The programs in the benchmark suite are written against an interface which is defined in a header file msg.h. This file in turn includes an implementation-specific header file called msg-impl.h. A configuration script will set up a symbolic link to an appropriate version. The mechanism permits that each function whose interface is defined in msg.h can either be defined as a macro in msg-impl.h (thus involving no run-time overhead at all) or as an external function. This form of configurability allows for efficient implementations with only minimal local interventions.

Since the low level interface introduces an extra thin layer, which mediates between applications and native communication libraries, its very existence might seem undesirable to implementers who wish to show their systems’ performances in the best possible light. However it is only by this kind of uniformity that code can be ported to new platforms without much hassle, and that results can be meaningfully compared. Furthermore, the approach is well in line with ATLAS’ larger strategy of giving preference to commercially supported hardware and software choices, where available.

By virtue of its design, the abstract interface can map onto MPI without any effort (where available) and to many vendor-specific low level communication libraries with only little effort (because their conventions often resemble MPI’s).

A precise definition of the API is given in [32]. It can be recapitulated as follows. There are 2 functions foreseen for preparing the library for initial use (corresponding to MPI_Init and for releasing its resources before program termination (MPI_Finalize). Two functions for enquiring the execution environment return information about the number of registered processes (MPI_Group_size) and the rank of the executing process (MPI_Comm_rank) respectively. Three pairs of binary functions enable non-blocking point-to-point communication (MPI_Isend, MPI_Irecv, MPI_Test and MPI_Wait in “symmetric” arrangements), 2 more enable blocking one-to-many communication (MPI_Comm_create and MPI_Bcast), and one performs barrier synchronization (MPI_Barrier). Finally, there is a function for suspending executing processes for a short time (provoked task switching).

Some simplifications apply to the number of function parameters as compared to standard MPI. Tags, communicators, and data types are not supported, mostly because their implementation incurs extra software overhead on performance. (This is for instance due to message dispatching on receivers’ sides and
to conversions between different byte-ordering schemes.) The associated additional semantics are only relevant or desirable in some applications. The ATLAS communication benchmarks do not belong to them.

MPI is a relatively large specification that offers \( \geq 150 \) functions to application programs. However, its device-dependent part is much smaller. This suggests that in a portable implementation the majority of MPI primitives can be expressed in terms of only few others that were chosen from a core set. (The usual trade-off between portability and efficiency clearly applies to how far this argument reaches as well.)

The MPICH reference implementation [31] takes this into account and defines an Abstract Device Interface (ADI), which can optionally be used for facilitating the porting of MPICH to new types of parallel systems (refer to section 3.3.1.1). The structure and expressive power of the MPICH ADI is similar to our low level interface, therefore the latter can conveniently serve to enable early implementations of full MPI for technologies which have already been tried with the ATLAS communication benchmarks.

### 3.2.2 Towards a possible one-copy implementation

The semantic gap between the message passing programming model, as represented by MPI, and a shared memory programming model, which for instance SCI [47] can enable in a distributed computing environment as well, has potentially important effects on achievable latencies and bandwidths. Processes that communicate over MPI assume the existence of multiple private and non-overlapping address spaces. Individual processes are therefore unprivileged to actively put or retrieve messages to or from mutually visible mailbox-like resources for exchange. (MPI-2 brings changes in this regard.) Instead, they passively relate
information about buffer locations in their private address spaces, when they issue send or receive requests to the MPI library. Communication is conceptually done by a privileged process that has a cluster-wide global view of shared resources. It copies messages on behalf of the communicating processes in between their private buffers.

In the following discussion we make several specific references to SCI (refer to section 3.3.4), which has appeared prominently as one particular candidate technology in our investigations. Nevertheless, similar observations can also be made for other manifestations of NUMA, such as Memory Channel [25, 26] and others.

For SCI the mentioned task requires that the message is written by the sender into an address space portion that has been mapped from SCI address space, and is therefore shared between 2 nodes. Because no general assumptions about where processes put their private buffers validly apply (they might for instance reside anywhere on the stack), an implementation may not assume that buffers already reside in a portion of the address space that is mapped as shared from a receiver. (SCI reserves local address spaces of up to $2^{48}$ addresses per node, which by definition excludes some addresses that a 64-bit CPU could, in principle, validly access.) Therefore, each point-to-point communication requires that the message is copied twice, which has undesirable effects on both latencies and bandwidths.

A proposal [33] has recently been made for potential use in the ATLAS LVL2 community. The new interface offers mainly improvements in 2 regards: it supports transparent routing in heterogeneous networks (of little concern to benchmarking), and it partly adopts the shared memory programming model. The latter is mainly for the benefit of technologies such as SCI and Memory Channel, because it allows for one-copy (or even zero-copy) instead of two-copy implementations. The invented scheme allows avoidance of the second copy, while still largely maintaining adherence to the message passing programming model. The latter is desirable mainly because of reasons that are listed in section 3.1.2.

The recent proposal requires a new API, whose specification was still in preparation when the measurements were performed (and more specifically when the ATLAS communication benchmarks were first precisely defined). Largely due to reasons of hardware availability the ATLAS communication benchmarks were initially mainly run on commercial multiprocessor systems and supercomputers. The reference implementation is therefore slightly biased towards message passing in general, and MPI in particular. Perhaps because of the lack of dynamic data structures and pointers in Fortran 77, MPI itself does not (yet) introduce similar shared-memory-oriented extensions. Maintaining equal bindings for both C and Fortran 77 was an important design requirement, which shaped the original MPI standard.

The alternative API effectively extends the old one, and can run on top of MPI as well. (Once MPI is used, performance gains are of course lost, for it maintains the paradigm with private address spaces and arbitrarily located buffers, which
unconditionally induces 2 copies per message transfer.)

In comparison, the new API accomplishes a strong integration of buffer management into the message passing layer. Therefore, it adds functions for allocation and deallocation of buffers to those that are directly concerned with communication. Senders and receivers must no longer use arbitrary memory resources from their private address spaces as communication buffers, but are required to collaborate with the communication library towards managing related resources. (The communication library in turn delegates buffer management to technology-specific sub-layers, for enabling transparent communication in heterogeneous clusters.)

According to the new scheme, both sending and receiving are achieved by pairs of binary communication primitives (the present form of presentation applies simplifications for better clarity): \texttt{mp_alloc} prepares a buffer for sending data; the implementation for SCI makes sure that all buffers reside in an address region that is already shared with the receiver over SCI. \texttt{mp_send} transmits the data and releases the buffer after completion; this now no longer needs to involve more than one physical data movement. (One-copy vs. two-copy implementation. Zero-copy implementations typically require shared memories with UMA characteristics.) \texttt{mp_recv} returns the location of a buffer where received data is already held. For SCI this buffer resides in shared memory as well, at an address that has previously been reported to the sender after a call to \texttt{mp_alloc}. \texttt{mp_free} releases this buffer after it has been emptied by application-level code on the receiver’s side. Both \texttt{mp_send} and \texttt{mp_recv} allow optional two-copy modes of operation which maintain downward compatibility to the present low level interface, and thus also to MPI.

### 3.3 Technology case studies

The following sub-sections deal with specific examples of communication fabrics, which are close to the leading edge among current technologies. Consequently, different groups in ATLAS presently consider them as appropriate candidate choices for the implementation of the networking part of a farm-based ATLAS LVL2 trigger (with the exception of the proprietary communication sub-system of the Meiko CS-2). The following discriminations can be drawn, in order to set the scene for more detailed presentations.

- The communication system of the Meiko CS-2 is one example out of many vendor-specific interconnects that serve in today’s parallel (super)computers. While a standardization effort in favor of RISC microprocessors has already brought their computational parts into close similarities, communication systems, such as those serving in the Meiko CS-2, IBM SP-2, or Cray T3E, remain largely in the domain of proprietary solutions (refer to section 3.1). This characteristic largely accounts for the systems’ higher per-node cost,
while still giving them a comfortable, if shrinking, performance edge over their competitors.

- ATM and Gigabit Ethernet (we will not discuss the latter in further detail) were chosen in ATLAS by mainly pragmatic reasoning. Their development is driven by large interest groups in the telecommunications and LAN markets, and both of them are covered by formal international standardization. For a successful technology in this area unit prices for related equipment are bound to decrease to mass-market levels. As intranets and the Internet continue to evolve around multimedia applications, a stable progression of offered bandwidths can be expected as well.

- SCI and Memory Channel were specifically designed with the aim in mind of interconnecting nodes in parallel computers. Conventional buses, which traditionally serve in this role, must be supplemented by structured communication networks as node counts increase (refer to section 3.1.1.2), in order to avoid costly communication bottlenecks. However, with parallel computing applications hardly overlapping with current mainstream market trends, it is unclear how these technologies will evolve in price.

### 3.3.1 The communication sub-system of the Meiko CS-2

The Meiko CS-2 is a parallel computer, which aims at aggregating the performance of dozens to hundreds of general-purpose SPARC microprocessors over a particular high-speed communication network. Machines with up to 256 processors have been manufactured and successfully delivered in the past. At the time of its installation in July 1994, CERN’s 64-node (128-processor) machine was the most powerful single computer system in the local computer center.

The machine’s nodes are boards with 1, 2, or 4 CPUs on a common SPARC Mbus shared memory interface, depending on the exact configuration. Vector processing capabilities can optionally be installed at individual nodes, for enhanced floating point performance.

Each node runs a copy of the Solaris 2.x operating system: it provides kernel-level support for multiprocessing with lightweight processes (Solaris threads). In the case of presence of more than one CPU on a Mbus interface, the operating system can make use of the pool of processing resources and separately schedule threads for execution on different CPUs, thus distributing the overall workload and allowing for speedup in multi-threaded programs and libraries.

This support of parallelism on the level of individual nodes is not Meiko-specific, but comes to the CS-2 essentially for free, because the machine’s CPU boards are designed for binary compatibility with Sun’s SPARC architecture: any program that runs on a SPARC workstation (under Solaris 2.x) will also run on the CS-2 without modifications. Important benefits are to be had from this
kinship, and the usual arguments that speak in favor of “open system design” apply. In this particular case vital software tools, such as command shells, editors, compilers, debuggers, libraries, and application packages, whose production from scratch would not be feasible for Meiko, are readily available because of Sun’s strong position as a supplier of workstations.

The CS-2 communication network [58] is largely specific, as is typically the case for today’s MPPs. Digital Equipment recently announced the use of similar technology from Quadrics in a new line of MPP systems.) Nodes are connected over 2 physically separate network layers with fat tree topologies. These layers are composed of individual 8 × 8 full crossbar switching elements. Each switching element is able to perform tasks relating to wormhole routing, CRC error checking, broadcast to a contiguous range of output links, and flow control in independent operation. It thereby causes an additional latency of ≤ 200 ns (according to the manufacturer).

Switching elements belong either to one network layer or the other, whereas nodes are connected to one switching element in each of the 2 layers. Fig. 3.11 shows the connections between nodes (rectangles) and switching elements (circles), which make up one network layer in a 64-node machine, such as the one that is presently installed at CERN. All connections are byte-wide and bidirectional (20 wires required for both directions), and they offer 50 MByte/s of usable bandwidth in each direction (70 MByte/s without protocol overheads).

Although applications with locality of reference in their communication behavior benefit from the hierarchic network design (messages in this case only have to traverse through lower branches of the tree), such locality is not strictly required. In logarithmic networks the worst case latency increases only slowly with the number of nodes, and in addition fat trees, unlike ordinary trees, feature constant bandwidth between stages. The three-dimensional view in Fig. 3.11

Figure 3.11: one layer of a three-stage CS-2 communication network

\[4\text{http://www.digital.com/PRW02T}\]
suggests the relative increase in bandwidth in stages that are closer to the root of the fat tree.

At the top stage there are as many connections out for expansion (not shown in the drawing) as there are nodes connected at the bottom stage; these connections can be used to conveniently double the size of the network without adding an extra stage. A fat tree’s bisection bandwidth grows linearly with the number of nodes; the topology is non-blocking, i.e. capable of supporting concurrent full bandwidth transfers between all pairs of nodes, and it offers multiple parallel routes between many nodes.

The existence of 2 independent network layers and of multiple redundant paths within each layer enhances bandwidth and adds fault tolerance to the communication network of the CS-2.

The connection between individual nodes and switching elements in the bottom stages of both network layers is via Elan Communication Processors (ECPs). One ECP is private to each node and acts as its gateway into the CS-2 communication network. An ECP is implemented using yet another SPARC processor on a node’s Mbus. The device deals exclusively with inter-node communication and prevents ongoing traffic from affecting a nodes’ main CPUs, in particular it relieves the main CPUs of the interrupt intensive operations associated with asynchronous inter-process communication. A special software support package allows system level reprogramming of ECPs.

By default, an ECP maintains routing tables where individual entries for each possible target identify up to 4 different paths. Based on this information, a sender’s ECP takes the full routing decision and predetermines the route via certain switching elements, at the time when a message is assembled, by providing a string of intermediate target addresses in the message header. The rationale for this is that by virtue of the ECP’s flexible design, which is largely reconfigurable under software control, as opposed to what is the case for the CS-2’s switching elements, ECPs allow updates of routing information more easily, in cases of machine reconconfigurations or partial failure.

Each switching element strips of a leading address from the header as it receives an incoming message and reveals the next intermediary address, before it takes its own routing decision. Thus addresses are appended by ECPs and cut off by switching elements. (This is similar to the header deletion mechanism on the C104 Asynchronous Packet Switch as described in section 4.3.2. Note Meiko’s past reputation as a supplier of transputer-based systems for a likely explanation of this similarity.)

The ECP essentially acts as communication coprocessor, which offers the functionality of a DMA engine that can act across the entire network. (The engine supports local DMA transfers equally as well, because communicating processes in a parallel program might very well execute on the same physical node.) Source and destination addresses are expressed in a two-dimensional logical address space, with node numbers and local virtual addresses as its 2 dimensions.
The ECP supports remote read, write, and synchronization operations; latency hiding is enabled by non-blocking communication instructions and can potentially be refined by reprogramming the embedded SPARC processor to closely match the requirements for higher communication protocol layers.

As a processor on the Mbus, the ECP has full access to a node’s shared memory; therefore it can implement the memory-to-memory send and receive primitives, which are required for the message passing programming model, in a straightforward manner. Since access privileges on remote addresses involved in DMA operations are checked by hardware on the ECP, user-level code can initiate remote memory accesses without having to make (expensive) system calls. (The overhead for single system call is \(\sim 10 - 100 \mu s\).) ECPs introduce message startup latencies of \(< 10 \mu s\) (according to the documentation).

The requirement for copying messages is removed on the level of communication libraries by the ability of the communications processor to operate in a node’s entire virtual store (it can in principle use the application’s own data structures as communication buffers, if the chosen synchronization mechanism allows for it) and on the level of application programs by the support of so called global objects.

The allocation of global objects is done such that at one time the same amount of storage is reserved in the memories of a set of nodes. Given correct allocation, global objects serve as data structures which are distributed over a set of processes and located at the same virtual address within each process. They are convenient, because they allow programs to make assumptions about the addresses of remote resources, such as communication buffers or events. Thereby they reduce the requirement for additional protocol steps and help to sustain high throughput rates. Global objects in Meiko’s terminology are similar to symmetric data objects on the Cray T3E in Cray’s terminology.

System support for inter-process message passing is available on the Meiko CS-2 on several levels of software abstraction: the Elan library [60] provides the lowest level functional interface to the ECP; it is where highest performance can be achieved and where other communication libraries are built on top. Unlike the Elan widget library [59], which is one abstraction layer above, it is not intended for direct use by applications. The Elan library covers the following areas.

- Elan events serve for synchronization between threads, such as those executing on a node’s main CPU and the ECP respectively. Elan events are to some extent similar to condition variables in Solaris and the POSIX threads standard [51]. The library offers functions for (re)setting events to a defined state, for polling and chaining them together, and for having a signal delivered to a process when an event is set, among others. The most common use of Elan events is for indicating that a DMA transfer has completed.

- The library includes functions to post requests for DMA transfers across the network. It queues requests, if necessary, and performs asynchronous,
reliable communication. The completion of a DMA operation can be flagged at either the sending or receiving end (or both) with Elan events.

- A mechanism for broadcasting to a consecutive range of destination processes is available.

The Elan widget library augments this functionality by service primitives for initialization, barrier synchronization, global exchange, allocation of global objects, handling of process groups (including support for non-contiguous sets), querying of the execution environment, exception handling, access to the parallel file system, and binary arithmetics. In addition, the library provides a threefold set of parallel programming constructs that are slightly more abstract than what is offered by the Elan library alone, and that address divergent application needs. Due to increasing software overheads, performance degrades slightly from DMA to Elan channels and Elan tports in turn. Elan channels offer the lowest latency message passing mechanism that is available on the CS-2.

- DMA support in the Elan widget library provides only a minimal wrapper around what is available in the Elan library already. It is best suited for bulk data transfers, where only little handshaking is required.

- Elan channels are similar (without implicit synchronization) to the constructs that Occam offers for implementing Hoare's communicating sequential processes [41]. Presumably their main purpose is to provide a migration path from transputer-related development tools for outdated Meiko equipment.

They provide full-duplex, bidirectional, unbuffered, and non-blocking message passing ports between pairs of processes. Because of the combination of the latter 2 features, both senders and receivers must have only one request for communication outstanding at each time. When a transmission completes, this also guarantees that a receive has been posted. Processes may keep any number of parallel connections open in between them, by using multiple channels. Channels for broadcasting are available in addition as well.

- Elan tports (tagged message ports) conform directly to the needs of popular message passing libraries. Tagged messages, non-blocking communication primitives, an arbitrary number of outstanding transmits and receives, and selection of incoming messages based on tags and senders correspond directly to matching features in MPI (and PVM).

Parallel programs are launched for execution on the CS-2 with the command `prun(1)`. These programs rely directly or through intermediate communication libraries (such as MPI) on the message passing functionality provided in the Elan
and Elan widget libraries. (Parallelism on the level of multiprocessor nodes, as expressed by multi-threading, is of no concern to \texttt{prun}(1), but is handled by the Solaris operating system alone. The shared memory programming model requires no explicit calls to communication primitives for communication, thus communication libraries are not required on this level, and thread-safe versions of standard libraries suffice.)

Typically, the nodes of a CS-2 system are arranged in a number of partitions. By allocating an appropriate number of nodes to each partition, the system administrator controls the type and amount of resources used for each class of work (such as interactive login, code development, and production runs), along with the associated access rights, scheduling policies, time limits, and accounting. (The nodes that were dedicated to executing parallel programs on the machine at CERN at the time when we did our benchmarks are drawn as gray rectangles in Fig. 3.11 on page 89).

The resource consumption of a parallel program is determined by the machine setup and by the set of command line options to \texttt{prun}(1).

\begin{verbatim}
prun -p partition -n \texttt{n_1} -N \texttt{n_2} program
\end{verbatim}

The command spawns \texttt{n_1} identical copies of the program onto \texttt{n_2} contiguous nodes in the specified partition. These instances form the so-called node segment. In addition, \texttt{prun}(1) always creates a host segment, which contains a single copy of a reserved process that forms the interface to the rest of the system during the life-time of the parallel program. (It provides e.g. line buffered output from all processes to the controlling terminal window.) The host segment typically executes in another partition, such as an interactive UNIX or batch partition. The CS-2 in principle supports multi-segment parallel programs, however \texttt{prun}(1) always uses only 2 segments.

\texttt{prun}(1) (by default) blocks execution, if it cannot immediately secure the required resources from \texttt{partition}. After resources have been granted, they remain reserved (by default) for the parallel program's exclusive use until \texttt{prun}(1) terminates, which is when all individual processes have exited, or until de-scheduling has been enforced by a higher-priority request. If \texttt{n_1} exceeds \texttt{n_2}, \texttt{prun}(1) schedules multiple processes to single nodes in a round-robin fashion. Apart from this, allocated processes have exclusive use of their nodes and don't have to share them with other user-level code.

3.3.1.1 An implementation of MPI for Elan channels and the MPICH channel interface

MPICH [30] is a complete and freely available implementation of the MPI specification. Its development proceeded with the very drafting of the standard itself. Because of its early availability and its ability to serve the conflicting goals of
high performance and straightforward portability well, MPICH serves as reference implementation, from which many vendors customized implementations were initially derived. Meiko’s MPI implementation for the Meiko CS-2, Digital MPI, and Sun MPI serve as examples in this regard.

MPICH aids portability with a layered software architecture, which defines an abstract device interface (ADI) at its core (refer to section 3.2.1). Vendors are encouraged to provide implementations of the ADI that produce excellent performance on specifically targeted machines, while taking advantage of the portability of the great majority of code above the ADI layer. The aim is to maximize the amount of code that can be shared without compromising performance.

In addition to support for a range of existing hardware, the MPICH distribution includes code that implements the ADI on top of several “virtual” devices. These correspond to older message passing libraries, such as Chameleon (the suffix -CH in MPICH in fact is derived from its name), and therefore conveniently enabled the earliest MPI implementations, by serving as bridges to trusted existing libraries.

The concept of “virtual” devices also provides opportunities for implementations of MPICH for an even lower cost than what corresponds to a full implementation of the ADI. One ADI implementation maps all functionality to the lower-level, yet still portable, MPICH channel interface [29], whose bare minimum requires a set of only 5 functions. Their combined semantics approximate the behavior of the system calls `read(2)`, `write(2)`, and `select(2)` in UNIX.

This software architecture allows for an incremental approach to trading portability for performance. The quickest way to port MPICH to a new environment is via the MPICH channel interface. Fig. 3.12 on the next page shows a protocol stack that can cluster a network of workstations via MPI, over readily available TCP/IP, and suggests an approximate correlation between groups of layers and the OSI reference model’s terminology. (The specific reference to a BSD kernel is of no general importance.)

The narrowing of the graphical representations of protocol layers from the MPICH channel interface downward suggests that effort can be saved by initially providing implementations at the level of this layer. The definition of the ADI as comprising a rich set of functions provides sufficient functionality for allowing efficient implementations of the protocol layers that reside on top, yet it also tolerates optional modules that emulate most of its functions in terms of the lower level MPICH channel interface. MPICH can thus be gradually tuned for a platform by replacing growing parts of shared code by platform-specific code (involving more development effort in exchange). These changes are transparent at the application layer, where users may benefit from the constant availability of well-defined MPI services.

Our reimplementation of MPICH for the Meiko CS-2 followed this agenda. The ensuing protocol stack is shown in the upper left corner of Fig. 3.13 on page 96. We chose Meiko’s Elan channel interface (no other relation than by
Figure 3.12: MPICH protocol stack and the OSI reference model

name to the MPICH channel interface) as basis on which to build on, in particular because it delivers higher performance than the Elan transport interface (refer to section 3.3.1).

Meiko’s own commercial implementation of MPICH opts for Elan tports, but makes no use of the MPICH channel interface layer, as is shown in the lower left corner of Fig. 3.13.

A high-performance implementation would attempt to combine both of these relative advantages, and thus implement the functions of the ADI interface directly in terms of the Elan channel interface, as is indicated in the upper right corner of Fig. 3.13. We nevertheless decided not to spend effort on an implementation of this type, because the ADI specification was scheduled to change towards an improved, but incompatible, version in releases after MPICH 1.0.12.

The ATLAS communication benchmarks can serve as a suitable test scenario for gaining evidence on the cost of using MPI instead of native communication libraries. By virtue of the low level interface, which has been modeled after a small subset of MPI, they can run in configurations involving either one of the sketched MPI implementations (left columns in Fig. 3.14) or a raw interface (right columns in Fig. 3.14). The latter is shown in the lower right corner of Fig. 3.13 for the arbitrarily highlighted case of the Elan channel interface.

Fig. 3.14 contains the following results: the minimal latency observed during inter-process communication is obtained from benchmark 1.1 (one-way) as \( t_s/2 \). The asymptotic bandwidth can be obtained from benchmark 1.4’s (pairs) quoted quantity \( \overline{t_r} \), as \( l/\overline{t_r} \), for large packet sizes.

There is room to argue that the ATLAS communication benchmark suite dis-favors configurations with MPI for the following reasons: MPI’s software overhead
Figure 3.13: protocol suites with and without MPICH-specific layers
<table>
<thead>
<tr>
<th></th>
<th>MPICH I/F</th>
<th>channel</th>
<th>ADI</th>
<th>—</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Elan I/F</td>
<td>channel</td>
<td>tport</td>
<td>channel</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.1</td>
<td>1.4</td>
<td>1.1</td>
</tr>
<tr>
<td>l</td>
<td></td>
<td>t_s</td>
<td>t_r</td>
<td>t_s</td>
</tr>
<tr>
<td>1</td>
<td>36</td>
<td>47</td>
<td>129</td>
<td>102</td>
</tr>
<tr>
<td>64</td>
<td>42</td>
<td>51</td>
<td>131</td>
<td>105</td>
</tr>
<tr>
<td>256</td>
<td>72</td>
<td>66</td>
<td>143</td>
<td>115</td>
</tr>
<tr>
<td>1024</td>
<td>158</td>
<td>132</td>
<td>196</td>
<td>164</td>
</tr>
</tbody>
</table>

Figure 3.14: results for Meiko CS-2 with and without MPICH-specific protocol layers

affects latency stronger than bandwidth. This is an embodiment of the general observation that bandwidth is mainly limited by the communication hardware, while latency is mainly limited by software overhead.

Since our measurements dealt only with packet sizes of up to 1 kByte, asymptotic bandwidths (where the effects of latency are minimal) were not approached. However, the particular parameter range was deliberately chosen with respect to the goals of our physics application; therefore the results are very well quantitatively meaningful within the given context. In addition, only limited effort went into the development of the implementation that was based on the MPICH channel interface. About 3 weeks were spent for familiarization with the MPICH software environment, coding, and debugging, which definitely leaves room for higher optimization.

The results indicate that there is a large performance gap between the raw Elan channel interface on one extreme end and MPI over the Elan tport interface on the other. In order to give a better grasp of this difference, Fig. 3.14 includes also results for a scenario where the ATLAS communication benchmarks exploited the raw Elan tport interface (not shown in Fig. 3.13). This suggests that the major part of the discrepancy ($\gtrsim 65 \mu s$ for lowest latencies) is due to different performance characteristics of the Elan channel and tport interfaces, which are intrinsic parts of Meiko’s system software. A smaller part ($\gtrsim 25 \mu s$ for lowest latencies) is effectively related to the usage of the indicated versions of MPICH over lower-level libraries.

### 3.3.2 ATM – Asynchronous Transfer Mode

ATM currently is at the focus of international research efforts with the aim of merging the communication infrastructures for high-speed transmission of voice,
video, and computer data through public and private networks with the future event of Broadband ISDN. At present, these services are supported by different (if partly overlapping) networks, such as telephone lines, corporate LANs, and cable television.

In the recent past the telephone network also had to support a part of the data traffic on the global Internet. As the number of users continues to grow, and the increase of available computing performance on individuals' desktops steadily enables new and ever more demanding applications (such as online videoconferencing and WebTV), industry looks forward to a new base-technology, which can handle different traffic types efficiently from its outset (for instance, voice and video signals have very different bandwidths requirements), and which promises a reliable path for implementing future growth. Moving from multiple networks to a single multi-service network, which integrates all telecommunication services and specialized networks into a common world-wide infrastructure, would also help to reduce cost and complexity of future installations.

Within the schematics proposed by the OSI network model ATM fits in roughly at the level of the data link layer. It is not strongly predisposed towards the use of a particular physical medium, because its designers foresaw that it would (at least initially) also reuse existing lines; thus ATM can run on top of different physical layers. Higher-level protocols, such as classical IP from the TCP/IP protocol's network layer, in particular, can make efficient use of ATM services as well.

In order to serve the needs of both connection-oriented services and data transmission well, ATM attempts to combine the advantages of both time division multiplexing (TDM) and packet switching. Its time slots are made available to connections on demand, thus a connection's allocated bandwidth is only filled up when packets are actually transmitted. The resulting framework is flexible enough to provide support for services that emphasize transmissions at constant delays, with guaranteed capacities, or for bursty traffic patterns over the same medium.

Data travels between ATM end-stations in the form of fixed-sized ATM cells, which contain 48 bytes of payload data prefixed with 5 bytes of header information. Their fixed size and format facilitates the implementation of high-speed switching fabrics.

The information in the header addresses the needs of routing, flow control, and error detection (only for the header, but not for the payload data). An ATM switch performs its function by inspecting the Virtual Path Identifier (VPI) and Virtual Channel Identifier (VCI) in an arriving ATM cell's header. It compares these addresses to entries in a private lookup table, takes a routing decision, and possibly replaces the VCI and VPI addresses, before it passes the ATM cell on to the next intermediate switch in the ATM network. ATM provides a connection-oriented service for a start; i.e., addresses used in the headers of ATM cells (relative to a segment between 2 ATM switches) are assigned for
complete durations of connections. (AAL4 enables also connection-less services over ATM.) Before connection establishment ATM end-stations are known by their ATM addresses, which are 20 bytes long.

The set of active connections consists of permanent virtual connections (PVCs), which are pre-set up at hardware installation time, and switched virtual connections (SVCs), which are only set up on demand by a signaling protocol. Each ATM end-station and the ATM network negotiate a contract, which is based on Quality of Service (QoS) parameters. The ATM device may e.g. use traffic shapers (such as de-randomizing buffers) to make sure that it adheres to its contract. The ATM network may for instance enforce QoS parameters by discarding cells from peers, which in turn do not obey their contractual obligations.

ATM proposes an ATM Adaption Layer (AAL), which translates between larger data units from higher protocol-layers, such as video streams or IP packets, and ATM cells. Its different aspects provide access to different service classes: AAL1 supports telephone and uncompressed video-traffic via constant bit-rate, real-time communication (“virtual wire”). AAL2 is still in the making; it supports e.g. compressed video-traffic via variable-bit-rate, real-time communication. AAL3/4 supports traffic patterns that are more typical for distributed computing workloads, via variable bit-rate, non-real-time communications, and for connection-oriented and connection-less operations respectively. AAL5 is a simpler and slimmer version of AAL3/4 and often serves in implementations of classical IP or LAN emulation over ATM.

The API to raw ATM services is typically an extension to the Berkeley socket interface. By adding support for an ATM address family, this enables a new inter-process communication domain, which utilizes raw ATM services rather than TCP/IP (or shared files). Digital UNIX, for instance, supports AAL5 over PVCs in its Native ATM Application Programming Interface for Digital UNIX 4.0.

Many of today’s commercially available ATM switches are mainly oriented towards the LAN market; they typically support maximum configurations with up to the order of dozens of ports. (16 × 16 in a given example.) Switches typically implement total capacities that provide sufficient bandwidth for fully non-blocking concurrent operation on all ports. An ATM switch with 5 GB/s capacity can thus e.g. support 32 ports with speeds of 155 MBit/s each (or a lower number of ports with higher port speeds, such as 622 MBit/s). The port speeds of 155 MBit/s and multiples thereof, which are typically associated with ATM, stem from the specifications of SONET and other physical layers.

Fig. 3.15 on the following page shows the results of first performance measurements of ping-pong latencies and bandwidths for a specific combination of commercial ATM network adapters and a commercial ATM switch, which were performed immediately after its installation in our lab at CERN. (These measurements were unrelated to the ATLAS communication benchmarks.)

The 2 adapters were ATMworks 350 155 MBit/s PCI cards from Digital
Equipment. These adapters can be used in combination with PCI-based workstations and servers; for the present tests 2 AlphaStations 200 4/166 under Digital UNIX 4.0B served as peers. Physical connectivity was established via multi-mode fiber-optic cables (SC connectors), which carried SONET/SDH-framed traffic. Once the connection was directly between the 2 devices and once via a ForeRunner ASX-1000 ATM backbone switch from Fore Systems.

The ASX-1000 switch offers ATM connectivity for up to 96 clients, with 10 GB/s of aggregate non-blocking bandwidth in its switching fabric. Each ATM port can run at port speeds between 1.5 MBit/s and 622 MBit/s, depending on the exact configuration. The manufacturer claims a switch transit delay of <11 μs, which is well in line with our measurements.

The Digital UNIX Native ATM Application Programmer’s Interface (ATMsock) was used as an abstraction for accessing ATM services in the benchmark programs. This API provides access to AAL5 services in a way that closely resembles the Berkeley socket interface to TCP/IP (and its datagram service in particular). The current version of ATMsock does only support PVCs, but not SVCs. However, this restriction is of no concern to our application, because connection setup times of <10 ms are prohibitive against dynamic connection establishment, in the first place. Two different VCs were used for in-bound and out-bound traffic on each ATM port. Various QoS parameters were left at their default values.

Ping-pong latencies were of the order of 65 – 75 μs (one way) for small messages (4 bytes). Fig. 3.15(a) shows the increase of latencies (both ways) as packet sizes increase. The difference between latencies as observed with or without the switch in between is ~10 μs, as can be readily expected from specs. The reason why the switch outperforms the direct connection for large packet sizes is cur-
rently not understood. If at all systematic, it is perhaps related to the interplay between ATM drivers and communication software that is executing inside of the ATM switch.

Fig. 3.15(b) on the facing page shows that the sustained asymptotic bandwidth is almost at 155 MBit/s; the remaining difference is due to the software overhead that is added by code in the benchmark programs and the ATM drivers and by the optional presence of the switch. (Digital claims that data rates over 134 MBit/s can be achieved at the user level, even with TCP/IP.) Half of the asymptotic bandwidth is reached at packet sizes of \( l_{1/2} \approx 1 \text{kByte} \).

### 3.3.3 Memory Channel

Memory Channel [25, 52] is a proprietary network technology from Digital Equipment Corporation, which is commercially targeted as inter-node transport medium in Digital UNIX TruCluster configurations. It typically interconnects several AlphaServers with multiple processors each, and thus extends the scalability of Digital’s product line to installations with currently up to 96 (= 8 \times 12), instead of 12 (maximum number of CPUs per UMA node), parallel Alpha processors.

This scalability will extend even further, when the current hub-based implementation for up to 8 ports will be updated to a full crossbar switch. Likewise, communication bandwidth is predicted to grow (the current generation of adapters uses FPGAs, which leaves room for future improvements), such that the interconnect can be driven near the limiting speed of the PCI bus (4\times33 \text{MByte/s theoretical peak for 32-bit PCI implementations, } \lesssim 100 \text{MByte/s observed in existing implementations}).

By virtue of their typically homogeneous hardware setup (one vendor) and operation at a single site (different appraisal of security issues), clusters allow the adoption of proprietarily optimized networking solutions. Thus they enable low-latency and high-bandwidth communication over a wide range of packet sizes and support fine-grained parallelism, by for instance abandoning "thick" protocol stacks such as TCP/IP. (The prevalence of numerous small messages often presents the worst traffic pattern for real-world networking technologies.)

Memory Channel claims to operate at one-way user-process-to-user-process latencies of \(< 5 \mu s\) for small messages with low-level, one-copy communication software [26]. Even if Digital MPI (an off-spring from the MPICH [30] reference implementation of the industry-standard message passing library) is employed, minimal observed latencies remain below the 10 \(\mu\)s barrier, according to the manufacturer. Sustained bandwidths for large packets (hundreds of kBytes in the case of Digital MPI) are reported as 64 MByte/s [52] and 61 MByte/s [52] respectively.

Since Memory Channel aims at reducing the software overhead in communication to a bare minimum, it is worth noting that its achieved performance
is in particular prominently dependent on the I/O configuration of the involved nodes.

Within a taxonomy of communication systems, MEMORY CHANNEL shares its approach with NUMAs (refer to section 3.1), in general, and with reflective memory systems, in particular. The SHRIMP (Scalable High-performance Really Inexpensive Multiprocessor) project’s SHRIMP-II adapter [8] for virtual memory-mapped communication specifically adopts a similar solution. The adapter conceptually works by snooping the sender’s PCI bus for write transactions to addresses that are mapped as shared with receiving nodes. Senders and receivers use calls to special library functions, which follow standard UNIX IPC calls in style, in order to register for communication. Possibly several receivers may participate in sharing overlapping parts of the virtual address space of a given sender; this enables multicast and broadcast traffic in a straightforward manner. The adapter puts intercepted data into low-level messages and sends them over the interconnect to the receivers’ network adapters, which signal cache invalidations to their hosts’ memory sub-systems (see Fig. 3.31 on page 123). Thus store instructions executed on one node are directly and instantly reflected in the physical memories of up to many other nodes.

Interconnects of this type attempt to reduce communication overheads by addressing several causes by a set of common means as further discussed in section 3.1.2.

In the particular case of MEMORY CHANNEL, the provision of communication services on the intra-node level is left to the operating system alone. Digital UNIX allows symmetric multiprocessing and includes support for several standardized software abstractions for shared memory access, such as the shmem (2) set of primitives from UNIX SVR4. However, these services cannot easily be applied to inter-node communication without modifications, partly because each (multiprocessor) node sees a different local address space and runs its own copy of the operating system.

Rather than combining all local address spaces, MEMORY CHANNEL provides a slightly different abstraction, namely an additional cluster-wide address space, which is 128 MByte in total size and is initially empty. This address space is subset in the PCI address space where the adapters logically reside, although this does not explicitly show above the software-driver level.

Different processes can populate the MEMORY CHANNEL address space by associating it partly with regions of their private virtual address spaces and thus with physical memory resources as well. These associations may be established as write-through or read-through, but must always remain unidirectional in the current implementation. The mappings’ granularity is fixed at the minimal Alpha page size of 8 kByte.

Portions of the MEMORY CHANNEL address space can be utilized for different purposes by using numeric keys; thus the set of valid keys imposes a permanent addressing scheme for cluster-wide MEMORY CHANNEL resources. Conceptually
speaking, a sequence of mappings, as shown in Fig. 3.16, mediates a partial overlap between the virtual address spaces of “distant” processes over the network. In these overlapping regions communication may be pursued.

Fig. 3.16 shows a particular snapshot from an example application with 2 nodes and 2 communicating processes executing on each of them. The 2 processes in the foreground may communicate bidirectionally with each other via 2 small buffers. The 2 processes in the background may only communicate from left to right via one mapped buffer.

This example shows that physical memory resources are only consumed at the receivers’ sides, where they are reserved and “pinned-down” by the first process that requests a given key. It also shows that processes may read and write to given regions in the MEMORY CHANNEL address space, but that for doing so, they have to establish a pair of unidirectional mappings and use different virtual addresses in the relevant load and store instructions. While software can be restructured from normal use such as to obey this asymmetry, such bidirectional mappings are generally undesirable performance-wise and otherwise.

MEMORY CHANNEL differs functionally from conventional shared-memory sub-systems also in several other regards, which are individually addressed below. Conventional shared memory (UMA) is e.g. used locally in AlphaServers with multiprocessor options installed. One could argue that for instance the lack of coherency on the cluster-wide level is in line with a general philosophy that is similar to arguments from the RISC vs. CISC discussion. These suggest to make explicit what the hardware can achieve best, to make the common case fastest and simple, and to rely on specialized software tools, such as optimizing compilers, for bridging ensuing semantic gaps and for enabling better results.
Asymmetry User processes that wish to communicate over MEMORY CHANNEL need to treat mapped regions as special. They may either write to or read from them, based on an early decision for one or the other. The decision has to be taken unrecoverably when the mapping is established. Reading and writing to single locations requires 2 mappings, and all pointers involved thus have to be represented as pairs of virtual address locations.

Application programs can be (re)written with moderate effort such that they use the appropriate address sibling for each type of access. This explicit treatment is nevertheless not always sufficient alone, because the compiler adds some semantic aspects autonomously, without obeying such “defensive” treatment of addresses. Accordingly, care must be taken not to use certain constructs, for otherwise the asymmetry can have peculiar effects. Auto-increment (or decrement) operators (on most CPUs) and access to non-aligned or byte-sized memory-locations (on most Alpha CPUs) will lead to segment violation errors, because RISC processors do not generally define atomic representations of these operations in their instruction sets. On some architectures, execution therefore leads to fatal read-modify-write sequences.

Early Alpha processors in particular followed a rather strict interpretation of the RISC philosophy of exposing hidden computations as a sequence of many simple and fast instructions. In view of some existing code bases this scheme has been recently revised; therefore the Alpha 21164 EV5.6 and Alpha 21164PC CPUs include byte and word manipulation extensions (BWX). Starting with Digital UNIX V4.0, BWX instructions can be emulated on all architectures; but for those that do not provide native support the necessity to use 32-bit longwords (or 64-bit quadwords) rather than bytes (or multiples thereof) in MEMORY CHANNEL address space remains, even in the presence of software emulation.

Read-write access to data structures in MEMORY CHANNEL address space should be avoided also for performance reasons, if possible. The network adapters copy data only to registered receivers, thus they do not effect changes in senders’ own memories by default. This is perhaps intended as an optimization that is favoring the most common case. But if another process on the same node (such as inc.mapper(1) or the very process that has executed the store instruction) needs read access to the same region as well, a special loopback feature must be enabled. As Fig. 3.17 on page 111 shows, this effectively reduces achievable bandwidth by about a factor of 2, because every out-bound write now additionally causes an in-bound read, and thus twice as much bandwidth is consumed on the PCI bus [52].

Coherency Memory regions that are shared between processes via the MEMORY CHANNEL address space are non-coherent; i.e., receivers must be aware that they may not gain access to the latest version of data items, as viewed by other nodes in the cluster, when they execute read instruction. (Read instructions
quasi behave like non-blocking read primitives in MPI parlance.) Only write instructions may initiate data transport over the network, while read instructions will return whatever data is available at present in local physical memories.

For mastering ensuing problems it is useful to distinguish between 2 latency aspects [19]. These are caused by the necessity to propagate signals between cluster nodes in a physical process that takes at least several microseconds (see Fig. 3.23 on page 116) and by the desire not to let fast processors stall while coherency is (re)established. (Contemporary, superscalar Alpha implementations with clock speeds ≤ 600 MHz and CPI ≥ 0.5 or 0.25 are theoretically capable of issuing thousands of new instructions during an equivalent amount of time.) Neither of these aspects needs explicit treatment in UMA or CC-NUMA shared memory systems, where a design choice for more complex memory sub-systems has been made.

**Initial coherency** describes that several processes may find a shared data structure in different initial states. The reason is that processes may request allocations at any time, and that the startup of some participating processes may be arbitrarily delayed. Regions in MEMORY CHANNEL address space may optionally be marked as coherent, for mastering initial coherency with a kind of startup synchronization. In this case a dedicated background process `imc_mapper(1)` immediately maps them for the benefit of other processes on the same node that may join later. A kernel-level RPC facility instructs its copies on remote nodes about necessary updates. Marking regions as coherent comes at a considerable cost, however, because all related writes into MEMORY CHANNEL address space become broadcasts, and because the loopback feature is now mandatory.

**Latency related coherency** reminds users that they have to plan explicit hand-shaking mechanisms between senders and receivers for distinguishing between data that are “merely” correct and have been delivered in the right order (this is what MEMORY CHANNEL accurately guarantees) and data that are most up-to-date (this remains in the responsibilities of applications or user-level communication libraries). Obviously this requirement must be met before applications can treat MEMORY CHANNEL address space as yet another form of shared memory. Flags, counters (updated after a sender deposits data, queried before a receiver considers them verbatim), or cluster-wide MEMORY CHANNEL locks (their use involves considerable overheads) are suitable candidate solutions.

Note that the familiar concept of coherency between a hierarchy of processors’ caches addresses a different architectural level, which almost only shares the use of similar terminology with the present discussion. MEMORY CHANNEL supports this kind of cache coherency, because writes from a sender will invalidate the relevant cache entries for mapped regions in receivers’ cache memories.
This conveniently allows receivers to operate busy spins on mapped data without destructively saturating the memory system.

**Weak ordering** Although MEMORY CHANNEL guarantees that data are delivered to remote nodes in the sequences in which they were written to its hardware, the Alpha architecture’s weakly ordered (rather than sequentially consistent) shared memory model may lead to a situation where the sender’s adapter observes stored data items in an order that differs from store instructions in the program code. Weak ordering imposes no implicit relation between the reads and writes issued on one processor, as viewed by a different processor (or by a network adapter for that matter), as a means for enabling better performance. A coherent view of shared data is suitably enforced whenever peers execute memory barrier assembly instructions (mb [18] or eieio [64]). Their semantics is such that all previous writes by a processor become visible to other processors, before subsequent writes show effects. In an arrangement where a flag or counter is set in shared memory in order to confirm that previously put data are up-to-date, these special instructions must be explicitly inserted between adjacent store instructions.

**Transparency** Although MEMORY CHANNEL introduces cluster-wide addressing, the difference between its address space and other address spaces is evident and requires explicit treatment outside the operating system kernel. The operating system bases its view of resources on the physical address space of the node where it is executing. Knowledge on how to convert between physical and virtual addresses via the processor’s address translation hardware is built into the kernel. Since MEMORY CHANNEL services are mostly implemented on a library level, the operating system cannot normally rely on MEMORY CHANNEL for providing basic services, such as rescheduling of threads between processors or balanced provision of other resources. (Digital UNIX’ Mach micro-kernel architecture imposes only light restrictions, in principle.) Because of this non-transparency, the external view of a cluster is different from that of a single node. Each node has its own IP address, for instance. Tools such as load levelers only partly hide the structural inhomogeneity. Thus for obtaining speedup while scaling beyond the size of a single node it is necessary to utilize special communication libraries to arrange for active scheduling operations from the user level (with protocols such as rsh(1)).

Digital supports access to MEMORY CHANNEL services on 4 different levels of software abstraction. The 2 lower levels are vendor-specific; the upper layers adopt industry-standard solutions.

- The MEMORY CHANNEL API library [19] resides at the lowest level and exposes a subset of the functionality of the kernel’s MEMORY CHANNEL sub-system to user processes. It follows the shared memory communication paradigm with functions that are partly analogous to the shmop(2) IPC
shared memory operations in UNIX SVR4 (see also Fig. 3.33 on page 125), and it introduces the smallest software overhead.

We chose it as suitable software layer for enabling the ATLAS communication benchmarks, because — according to the envisaged needs of our target application in the domain of the ATLAS LVL2 trigger — these benchmarks focus on highlighting systems’ abilities to achieve minimal latencies and efficient transmissions of numerous small to moderately sized messages.

The functions provided by the MEMORY CHANNEL API library can be categorized as follows: initialization and finalization, mapping between virtual address spaces and MEMORY CHANNEL address space, provision of cluster-wide locks (similar to mutexes in the POSIX threads interface standard), and miscellaneous services (environment enquiry, delivery of signals to remote nodes, error counting, and reporting).

- UMP (Universal Message Passing) is a proprietary message passing library, which Digital provides as running on top of the MEMORY CHANNEL API library. Its function is mainly twofold: to provide a uniform abstraction over both conventional shared memory (inside nodes) and memory in MEMORY CHANNEL address space (inside clusters), for the benefit of user-level code that wishes to use one or the other in a transparent fashion, and to enable message passing on top of the shared memory programming model.

  Digital has arranged its own communication middleware modules as UMP’s clients. But use of the library by outside parties is currently not encouraged (and documentation is sparse), which is why we did not consider it as part of the protocol stack for enabling the ATLAS communication benchmarks.

- Higher-level communication libraries include Digital MPI, Digital PVM, and the run-time system of High Performance Fortran. Their emphases are mainly on portability and reliability, which causes different trade-offs concerning achievable latencies and bandwidths.

  They give best results when messages are large (asymptotic bandwidths are approached), when a coarse form of parallelism is used (longer latencies are then less of an issue, because communication can readily overlap with computation), or when code has to be developed on non-production platforms with only compatible software arrangements (such as networks of workstations).

  Because of their relative heavy-weightiness (code for the MPICH reference implementation is likely to remain much larger than the executable ATLAS LVL2 trigger code), we did not consider any of these members as a prime choice for studying ATLAS communication benchmarks. Results for Digital MPI are nevertheless included, since they can be achieved with only minimal extra effort (refer to section 3.2.1), and since many large computer systems
(such as the Cray T3E, for instance) initially allowed only for MPI-based results for the ATLAS communication benchmarks. Thus their provision permits more straightforward and encompassing comparisons.

• Digital UNIX includes a TCP/IP network driver for Memory Channel. Hence all programs from the application layer of the TCP/IP protocol stack can be instantly applied. (This added some convenience and for instance enabled the first set of results, because it allowed the use of \texttt{telnet(1)} over Memory Channel, while some Ethernet interfaces were down.)

Among the large code base that already uses TCP/IP for communication are e.g. NFS and distributed (parallel) file systems. Large database applications, for instance, can profit from multiprocessor servers with cluster configurations, which can scale widely without necessitating expensive software migration tasks.

Fig. 3.18 on page 112 shows the resources that the technology-specific implementation of the low level interface acquires. $4+n^2+n$ numeric keys are assigned in total, where $n$ is the number of communicating processes.

Three sets of Memory Channel locks are required for performing barrier synchronization at startup, for resolving initial coherency, and for establishing process ranks respectively. The latter task is accomplished by applying the following strategy: each process allocates a set of $n$ locks, and immediately tries to acquire one of them by looping over all members. The index of the first lock that a process gets determines its rank. Processes then hold these locks until program termination.

$n \times n$ entries in the global Memory Channel address space reserve buffer space for bidirectional point-to-point communication between every pair of processes. Processes map only those entries that enable them to exchange data with their ”neighbors”. The ensuing directionality of mappings (read or write) is indicated by differently oriented black triangles in Fig. 3.18 on page 112. In addition, each process also maps $n$ buffers for broadcast communication.

Each field in the matrix has room for an entire queue of messages that were posted by senders, but were not yet extracted by receivers. The static organization of the queue following Fig. 3.19(a) on page 112 imposes restrictions on the size of messages and on the maximum number of outstanding messages between each pair of processes; a more flexible implementation could easily overcome these limitations. The extra queue element that is drawn at a slight offset in Fig. 3.19(a) is semi-redundant: it helps to distinguish between full and empty queues by evaluating equivalence of $\texttt{b.head}$ and $\texttt{f.tail}$. Head and tail indices that semantically belong to the same queue are syntactically distributed to 2 structures as corresponding to 2 adjacent fields relative to the matrix diagonal in Fig. 3.18. This arrangement is recorded by the use of different prefixes for denoting the 2 structures in our notation. It is necessary in face of the unidirectionality
of Memory Channel mappings and the fact that head and tail indices needs incrementation (write access) by receivers and senders respectively.

A further complication in relation to the static queue arises when an implementation of the one-copy message passing library, following the outline that is given in section 3.2.2, is desired. Reading out and freeing queue elements thus become related to 2 different calls, whose calling sequences can be arbitrarily intermixed for different messages. Fig. 3.19(b) on page 112 sketches a possible data structure that accounts for this asynchronicity by effectively maintaining both the queue and a free-list in a single array.

The presently discussed implementation of the low level interface can be configured at compile-time, by manipulating the following set of C preprocessor macros.

**BASE_KEY** The implementation requires that all keys in the range \( k_0 \leq k < k_0 + 4 + n^2 + n \) are reserved for its private and exclusive use, where \( k_0 \) is the numeric value that is associated with `BASE_KEY` (see Fig. 3.18 on page 112). Its default value has been randomly chosen as 4000.

**CHECKSUM_MODE** If this macro is defined, calls to `msg_send` and `msg_mcast` pass checksum information, along with the very data, to receivers. Receivers verbosely complain, if they detect checksum errors. The option has been included as a means to facilitate software debugging. The Memory Channel API library offers more appropriate features for detecting hardware errors by itself.

**MAX_GROUP_SIZE** The numeric value of this macro controls the maximum size of process groups that can be passed in calls to `msg_bcast`.

**MAX_QUEUE_LEN** The numeric value of this macro defines the maximum number of outstanding messages between each pair of senders and receivers. When exceeding this count, the semantics of `msg_send` turns into the equivalent of a blocking call.

**MC_UNIT** Digital UNIX TruCluster installations allow up to several Memory Channel adapters per node, for enhanced network availability and throughput. Our implementation uses exactly one adapter, whose index is given by the numeric value of the macro.

**RANDOM_MODE** The communication library inserts random test patterns as message contents, if this macro is defined. It facilitates software debugging, if `CHECKSUM_MODE` is enabled at the same time as well. Only those benchmarks that do not depend on uncompromised message contents (1.x and 3.x) tolerate this setting.
**SINGLE_NODE_MODE** If this macro is defined, the implementation assumes that at least 2 processes that participate in the ATLAS communication benchmarks are executing on the same node; it then chooses conservative optimization options. In particular, it inserts calls to `usleep(3)` into perpetual loops, in order to enforce frequent task switchings, and it allocates/maps all portions from MEMORY CHANNEL address spaces with flags governing coherency and loopback enabled.

**VERBOSE_MODE** This macro supports debugging by controlling the output of various diagnostic messages. It is turned off by default.

### 3.3.3.1 Maximum sustained bandwidth without handshake

Fig. 3.17 on the facing page shows maximum bandwidths that can be achieved for writes into MEMORY CHANNEL address space (without congestion). These results are well in line with the peak bandwidth of $\sim 64$ MByte/s, which Digital quotes e.g. in [52].

The plots show that the raw CPU speed is not the limiting factor for any of the nodes that we put to test. All 3 types can saturate MEMORY CHANNEL almost equally well. The slightly higher rise of the top curve in Fig. 3.17(a) is perhaps due to the AlphaServer’s more sophisticated PCI bus implementation, which helps to increase its asymptotic bandwidth.

The initial sharp peak for 2 of the 3 types of nodes (those with an Alpha 21164 instead of an Alpha 21064 processor, as it turns out) at a packet size of 32 bytes may either be non-systematic or is perhaps attributable to effects from internal caching. (32 bytes corresponds to the size of single cache lines.)

These results show that, unlike for writes into local DRAM memory, MEMORY CHANNEL does not seem to profit from memory accesses in units of quadwords (64 bits) rather than longwords (32 bits). As representatives of a true 64-bit processor architecture, Alpha CPUs can perform operations on both granularities with the same efforts. The present generation of MEMORY CHANNEL cards is tailored to the 32-bit version of PCI. Therefore, the highest possible speed of the network adapter seems to prevail as limiting over what the CPU can deliver to such a prominent extent that the network can be fully saturated, even if the CPU has to go through twice as many instructions in order to copy out blocks of data.

What does make a large impact, however, is whether transmit regions are attached with or without loopback. In loopback mode all writes into MEMORY CHANNEL address space are reflected back to the originating node’s memory; thus each transaction consumes twice as much bandwidth on the PCI bus.
Figure 3.17: maximum sustained bandwidths for writes into Memory Channel (the graphs plot packet sizes in bytes vs. observed durations in $\mu$s)
Figure 3.18: overview of cluster-wide shared data structures

Figure 3.19: schematic view of synchronous and asynchronous queues
3.3.3.2 Results for ATLAS communication benchmarks with and without Digital MPI

The results were obtained with the following experimental setup: one AlphaServer 4000 5/300 (299 MHz Alpha 21164 E5, 96 kByte + 2 MByte off-chip cache) and 4 AlphaStations 200 4/166 (166 MHz Alpha 21064 EV4.5, 512 kByte off-chip cache) were combined in a Digital UNIX TruCluster. Inter-node connectivity was established over MEMORY CHANNEL adapter cards (revision 1.5), a MEMORY CHANNEL hub with 5 line cards, and copper link cables.

For some brief time we also had an AlphaStation 500/400 (400 MHz Alpha 21164A E5.6, 96 kByte + 2 MByte off-chip cache) available as an intermediary replacement for another node.

All workstations had 128 MByte RAM configurations and ran copies of the Digital UNIX 4.0B operating system. MPI-related tests were done with field trial versions 1.0 of Digital MPI for MEMORY CHANNEL clusters (with v4patch.tar installed). The MEMORY CHANNEL API [19] came as part of the Digital TruCluster Software Version 1.4. Although at present this software is only commercially supported for AlphaServers (machines that typically come with symmetric multi-processor options installed), the fact that both AlphaServers and AlphaStations rely on a similar PCI-based technology allowed a quick path for enabling MEMORY CHANNEL also on some workstation types. (This evidently did not work for
the AlphaStation 255/233.)

Among these processors the AlphaServer 4000 5/300 implemented the most sophisticated I/O sub-system (64-bit PCI implementation, 2 PCI buses), as can be expected from its market positioning and from typical server workloads. All nodes used identical MEMORY CHANNEL adapters; thus the server’s 64-bit capability was not specifically exploited.

In terms of CPU speed, the server ranked only second behind the more recent AlphaStation 500/400. Each line of Fig. 3.20 shows the relative performance of all available nodes. The upper and lower numbers in each box quote SPECint95 and SPECfp95 ratings respectively. These numbers show that the AlphaStation 500/400 outperforms the server by about 50% and the slower stations by a factor of \(~4\) on integer-computing-intensive tasks.

We used 2 to 5 cluster nodes, were each node executed one copy of a communicating process that was specific to the tried benchmark. The cluster was dedicated to the benchmarks at all relevant times; thus there was no interference from other workloads other than from basic operating system services.

By controlled scheduling of processes for execution on specific nodes we were able to study the effects of different processor hardware (while adapter hardware remained) on the benchmarking results. The set of cluster configurations that is shown in Fig. 3.20 is meant to reveal peak and worst-case performances (especially on round-trip latencies for small messages) and to include the particular configuration that will remain in our lab for further production, after the abstract communication benchmarks have been completed. In the case of Digital MPI, for instance, adding and changing between configurations is merely a matter of specifying a different host file in calls to \texttt{mpirun(1)} (see also Fig. 3.9 on page 83).

configuration 1 This balanced client-server configuration will remain available for future use. The server is listed first; thus it takes part in all benchmarks. In asymmetric communication topologies, like in the benchmarks outfarming (1.5) to funnel (1.7), it executes the most communication/computation intensive task. The remaining cluster nodes are from a homogeneous pool of slower workstations. Round-trip latencies in benchmark one-way (1.1) and two-way (1.2) are measured between the server and one of its clients.

configuration 2 The purpose of this configuration is to show to what extent round-trip latencies increase as a function of processor performance. Round-trip latencies are measured between the 2 fastest nodes. Some speed-up can be expected for the other benchmarks as well, because the original server was replaced by an even faster node for the most time-critical processes.

configuration 3 This configuration serves to manifest worst-case round-trip latencies, as measured between slow clients. This is not a sensible configuration for production, because the faster nodes have been shifted into positions of relative obscurity in the pool of clients.
**Figure 3.21:** results for configuration 1 (all times in µs, $l_{\text{min}} = 8$)

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n_s$</th>
<th>$n_w$</th>
<th>$n_r$</th>
<th>$l$</th>
<th>$t_s$</th>
<th>$T$</th>
<th>$T_r$</th>
<th>$T_{\text{f}}$</th>
<th>$T_{\text{u}}$</th>
<th>$T_{\text{u}}$</th>
<th>$T_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>12.9</td>
<td>8.1</td>
<td>8.9</td>
<td>3.9</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>64</td>
<td>18.3</td>
<td>13.1</td>
<td>13.3</td>
<td>9.6</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>256</td>
<td>37.9</td>
<td>34.4</td>
<td>34.1</td>
<td>22.9</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1024</td>
<td>118.3</td>
<td>122.6</td>
<td>120.9</td>
<td>81.6</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>—</td>
<td>—</td>
<td>4.8</td>
<td>8.4</td>
<td>4.6</td>
<td>26.6</td>
<td>36.6</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>64</td>
<td>—</td>
<td>—</td>
<td>9.2</td>
<td>12.4</td>
<td>8.6</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>256</td>
<td>—</td>
<td>—</td>
<td>22.9</td>
<td>23.8</td>
<td>26.1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1024</td>
<td>—</td>
<td>—</td>
<td>84.0</td>
<td>80.9</td>
<td>95.3</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>—</td>
<td>—</td>
<td>20.2</td>
<td>4.2</td>
<td>—</td>
<td>29.9</td>
<td>39.9</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>64</td>
<td>—</td>
<td>—</td>
<td>45.5</td>
<td>8.9</td>
<td>—</td>
<td>33.3</td>
<td>53.3</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>256</td>
<td>—</td>
<td>—</td>
<td>111.1</td>
<td>24.4</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1024</td>
<td>—</td>
<td>—</td>
<td>379.9</td>
<td>88.1</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>—</td>
<td>—</td>
<td>9.7</td>
<td>9.9</td>
<td>9.4</td>
<td>33.3</td>
<td>46.6</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>64</td>
<td>—</td>
<td>—</td>
<td>15.2</td>
<td>13.9</td>
<td>12.8</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>256</td>
<td>—</td>
<td>—</td>
<td>38.7</td>
<td>26.1</td>
<td>35.3</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1024</td>
<td>—</td>
<td>—</td>
<td>170.4</td>
<td>82.1</td>
<td>129.6</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

**Figure 3.22:** results for configuration 1 with Digital MPI (all times in µs, $l_{\text{min}} = 1$)
Figure 3.23: results for configuration 2 (all times in μs, $l_{\text{min}} = 8$)

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n_s$</th>
<th>$n_w$</th>
<th>$n_r$</th>
<th>$l$</th>
<th>$t_s$</th>
<th>$\bar{t}$</th>
<th>$\bar{r}_r$</th>
<th>$\bar{r}_r$</th>
<th>$t_r$</th>
<th>$t_u$</th>
<th>$t_u$</th>
<th>$t_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>8</td>
<td>10.8</td>
<td>6.9</td>
<td>7.1</td>
<td>2.3</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>64</td>
<td>14.8</td>
<td>9.4</td>
<td>9.8</td>
<td>3.8</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>256</td>
<td>31.1</td>
<td>24.0</td>
<td>24.4</td>
<td>9.9</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1024</td>
<td>96.6</td>
<td>81.8</td>
<td>82.6</td>
<td>46.4</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

| 3   | 1     | 1     | 1     | —   | 8     | —     | —         | —         | 4.9   | 7.3   | 5.3   | 23.3  | 33.3  | 22.8  |
| 3   | —     | —     | —     | —   | 64    | —     | —         | —         | 8.9   | 11.0  | 8.8   | —     | —     | —     |
| 3   | —     | —     | —     | —   | 256   | —     | —         | —         | 23.2  | 21.6  | 25.8  | —     | —     | —     |
| 3   | —     | —     | —     | —   | 1024  | —     | —         | —         | 94.2  | 79.3  | 94.8  | —     | —     | —     |

| 4   | 1     | 1     | 2     | —   | 8     | —     | —         | —         | 29.2  | 3.9   | —     | —     | 23.3  | 33.3  | 11.8  |
| 4   | 2     | 1     | 1     | —   | 64    | —     | —         | —         | 45.4  | 6.9   | —     | —     | 29.9  | 43.3  | 9.2   |
| 4   | —     | —     | —     | —   | 256   | —     | —         | —         | 114.1 | 19.0  | —     | —     | —     | —     | —     |
| 4   | —     | —     | —     | —   | 1024  | —     | —         | —         | 398.4 | 70.8  | —     | —     | —     | —     | —     |

| 5   | 2     | 1     | 2     | —   | 8     | —     | —         | —         | 9.9   | 9.4   | 10.8  | 33.3  | 46.6  | 9.3   |
| 5   | 1     | 2     | 1     | —   | 64    | —     | —         | —         | 15.8  | 12.7  | 13.4  | —     | —     | —     | —     |
| 5   | —     | —     | —     | —   | 256   | —     | —         | —         | 40.4  | 23.8  | 33.6  | —     | —     | —     | —     |
| 5   | —     | —     | —     | —   | 1024  | —     | —         | —         | 189.0 | 80.2  | 125.1 | —     | —     | —     | —     |

Figure 3.24: results for configuration 2 with Digital MPI (all times in μs, $l_{\text{min}} = 1$)

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n_s$</th>
<th>$n_w$</th>
<th>$n_r$</th>
<th>$l$</th>
<th>$t_s$</th>
<th>$\bar{t}$</th>
<th>$\bar{r}_r$</th>
<th>$\bar{r}_r$</th>
<th>$t_r$</th>
<th>$t_u$</th>
<th>$t_u$</th>
<th>$t_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>40.8</td>
<td>27.4</td>
<td>29.5</td>
<td>13.6</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>8</td>
<td>27.9</td>
<td>17.4</td>
<td>18.7</td>
<td>6.4</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>64</td>
<td>34.1</td>
<td>21.7</td>
<td>25.5</td>
<td>8.4</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>256</td>
<td>50.1</td>
<td>39.9</td>
<td>40.9</td>
<td>16.4</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1024</td>
<td>123.1</td>
<td>107.9</td>
<td>110.9</td>
<td>53.1</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

| 3   | 1     | 1     | 1     | —   | 1     | —     | —         | —         | 62.3  | 52.4  | —     | —     | 353.3 | 399.9 | 52.8  |
| 3   | —     | —     | —     | —   | 8     | —     | —         | —         | 27.3  | 16.3  | —     | —     | —     | —     | —     |
| 3   | —     | —     | —     | —   | 64    | —     | —         | —         | 32.2  | 22.4  | —     | —     | —     | —     | —     |
| 3   | —     | —     | —     | —   | 256   | —     | —         | —         | 51.9  | 40.2  | —     | —     | —     | —     | —     |
| 3   | —     | —     | —     | —   | 1024  | —     | —         | —         | 161.4 | 160.5 | —     | —     | —     | —     | —     |

| 4   | 1     | 1     | 2     | —   | 1     | —     | —         | —         | 427.3 | 33.4  | —     | —     | 349.9 | 439.9 | 47.1  |
| 4   | 2     | 1     | 1     | —   | 8     | —     | —         | —         | 218.0 | —     | —     | —     | 376.6 | 486.6 | 74.3  |
| 4   | —     | —     | —     | —   | 64    | —     | —         | —         | 278.4 | —     | —     | —     | —     | —     | —     |
| 4   | —     | —     | —     | —   | 256   | —     | —         | —         | 356.4 | —     | —     | —     | —     | —     | —     |
| 4   | —     | —     | —     | —   | 1024  | —     | —         | —         | 719.5 | —     | —     | —     | —     | —     | —     |

<p>| 5   | 2     | 1     | 2     | —   | 1     | —     | —         | —         | 64.9  | 99.5  | —     | —     | 419.9 | 536.6 | 74.6  |
| 5   | 1     | 2     | 1     | —   | 8     | —     | —         | —         | 42.8  | 35.4  | —     | —     | 58.6  | —     | —     |
| 5   | —     | —     | —     | —   | 64    | —     | —         | —         | 39.7  | 41.8  | —     | —     | —     | —     | —     |
| 5   | —     | —     | —     | —   | 256   | —     | —         | —         | 68.6  | 66.2  | —     | —     | —     | —     | —     |
| 5   | —     | —     | —     | —   | 1024  | —     | —         | —         | 230.8 | 248.4 | —     | —     | —     | —     | —     |</p>
<table>
<thead>
<tr>
<th>( n )</th>
<th>( n_s )</th>
<th>( n_w )</th>
<th>( n_r )</th>
<th>( l )</th>
<th>( t_s )</th>
<th>( r )</th>
<th>( t_r )</th>
<th>( t_r )</th>
<th>( t_r )</th>
<th>( t_u )</th>
<th>( t_u )</th>
<th>( t_r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>8</td>
<td>15.4</td>
<td>9.3</td>
<td>10.2</td>
<td>4.1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>64</td>
<td>23.8</td>
<td>15.6</td>
<td>15.9</td>
<td>8.6</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>256</td>
<td>55.4</td>
<td>40.8</td>
<td>40.8</td>
<td>24.9</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1024</td>
<td>185.1</td>
<td>140.0</td>
<td>139.5</td>
<td>93.9</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>8</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>9.3</td>
<td>9.6</td>
<td>8.8</td>
<td>29.9</td>
<td>43.3</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>64</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>17.3</td>
<td>15.5</td>
<td>15.1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>256</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>50.4</td>
<td>32.9</td>
<td>45.8</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1024</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>190.6</td>
<td>104.1</td>
<td>174.8</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>—</td>
<td>—</td>
<td>27.6</td>
<td>4.2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>33.3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>64</td>
<td>—</td>
<td>—</td>
<td>42.1</td>
<td>8.7</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>29.9</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>256</td>
<td>—</td>
<td>—</td>
<td>107.8</td>
<td>25.6</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1024</td>
<td>—</td>
<td>—</td>
<td>376.6</td>
<td>94.8</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>17.2</td>
<td>11.0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>64</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>32.5</td>
<td>16.3</td>
<td>32.9</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>256</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>100.7</td>
<td>34.5</td>
<td>99.6</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1024</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>379.2</td>
<td>106.2</td>
<td>372.4</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Figure 3.25: results for configuration 3 (all times in \( \mu s \), \( l_{\min} = 8 \))

<table>
<thead>
<tr>
<th>( n )</th>
<th>( n_s )</th>
<th>( n_w )</th>
<th>( n_r )</th>
<th>( l )</th>
<th>( t_s )</th>
<th>( r )</th>
<th>( t_r )</th>
<th>( t_r )</th>
<th>( t_r )</th>
<th>( t_u )</th>
<th>( t_u )</th>
<th>( t_r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>150.6</td>
<td>121.4</td>
<td>129.8</td>
<td>54.8</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>8</td>
<td>74.8</td>
<td>52.2</td>
<td>57.8</td>
<td>25.3</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>64</td>
<td>84.4</td>
<td>59.9</td>
<td>70.9</td>
<td>29.1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>256</td>
<td>125.3</td>
<td>93.1</td>
<td>92.4</td>
<td>46.1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1024</td>
<td>264.9</td>
<td>199.8</td>
<td>207.2</td>
<td>139.3</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>96.8</td>
<td>91.1</td>
<td>•</td>
<td>383.3</td>
<td>476.6</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>8</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>32.4</td>
<td>24.4</td>
<td>•</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>64</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>43.5</td>
<td>31.9</td>
<td>•</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>256</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>79.7</td>
<td>58.3</td>
<td>•</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1024</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>267.9</td>
<td>189.7</td>
<td>•</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>416.1</td>
<td>55.5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>379.9</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>—</td>
<td>—</td>
<td>236.9</td>
<td>25.4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>353.3</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>64</td>
<td>—</td>
<td>—</td>
<td>270.7</td>
<td>30.5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>256</td>
<td>—</td>
<td>—</td>
<td>364.4</td>
<td>48.6</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1024</td>
<td>—</td>
<td>—</td>
<td>871.4</td>
<td>136.9</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>211.0</td>
<td>144.9</td>
<td>•</td>
<td>376.6</td>
<td>439.9</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>64.2</td>
<td>35.8</td>
<td>•</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>64</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>87.4</td>
<td>48.4</td>
<td>•</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>256</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>163.5</td>
<td>87.3</td>
<td>•</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1024</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>494.2</td>
<td>283.6</td>
<td>•</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Figure 3.26: results for configuration 3 with Digital MPI (all times in \( \mu s \), \( l_{\min} = 1 \))
### Table 3.27: Results for Configuration 4 (all times in $\mu$s, $l_{\text{min}} = 8$)

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n_s$</th>
<th>$n_w$</th>
<th>$n_r$</th>
<th>$l$</th>
<th>$t_s$</th>
<th>$t_l$</th>
<th>$t_r$</th>
<th>$t_r$</th>
<th>$t_r$</th>
<th>$t_r$</th>
<th>$t_r$</th>
<th>$t_r$</th>
<th>$t_r$</th>
<th>$t_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1024</td>
<td>118.6</td>
<td>122.1</td>
<td>121.2</td>
<td>81.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>12.9</td>
<td>8.1</td>
<td>8.9</td>
<td>3.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>64</td>
<td>18.1</td>
<td>13.1</td>
<td>13.4</td>
<td>9.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>256</td>
<td>38.1</td>
<td>34.4</td>
<td>34.1</td>
<td>22.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1024</td>
<td>366.6</td>
<td>366.6</td>
<td>223</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>28.7</td>
<td>4.2</td>
<td></td>
<td></td>
<td>26.6</td>
<td>39.9</td>
<td>15.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>64</td>
<td>44.2</td>
<td>9.1</td>
<td></td>
<td></td>
<td>33.3</td>
<td>46.6</td>
<td>10.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>256</td>
<td>109.6</td>
<td>24.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1024</td>
<td>380.6</td>
<td>88.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>9.5</td>
<td>9.4</td>
<td>9.4</td>
<td>33.3</td>
<td>49.9</td>
<td>10.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>64</td>
<td>15.2</td>
<td>13.6</td>
<td>13.3</td>
<td></td>
<td></td>
<td></td>
<td>20.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>256</td>
<td>38.9</td>
<td>25.1</td>
<td>35.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1024</td>
<td>168.6</td>
<td>80.5</td>
<td>129.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 3.28: Results for Configuration 4 with Digital MPI (all times in $\mu$s, $l_{\text{min}} = 1$)

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n_s$</th>
<th>$n_w$</th>
<th>$n_r$</th>
<th>$l$</th>
<th>$t_s$</th>
<th>$t_l$</th>
<th>$t_r$</th>
<th>$t_r$</th>
<th>$t_r$</th>
<th>$t_r$</th>
<th>$t_r$</th>
<th>$t_r$</th>
<th>$t_r$</th>
<th>$t_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>119.4</td>
<td>114.3</td>
<td>114.8</td>
<td>54.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>8</td>
<td>54.1</td>
<td>48.4</td>
<td>33.6</td>
<td>23.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>64</td>
<td>66.6</td>
<td>59.0</td>
<td>60.9</td>
<td>27.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>256</td>
<td>92.8</td>
<td>87.8</td>
<td>86.4</td>
<td>43.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1024</td>
<td>186.4</td>
<td>201.7</td>
<td>195.4</td>
<td>187.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>58.4</td>
<td>53.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>8</td>
<td>26.3</td>
<td>17.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>64</td>
<td>31.4</td>
<td>21.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>256</td>
<td>49.8</td>
<td>39.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1024</td>
<td>140.0</td>
<td>130.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>306.1</td>
<td>56.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>214.4</td>
<td>24.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>64</td>
<td>225.2</td>
<td>29.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>256</td>
<td>319.9</td>
<td>46.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1024</td>
<td>630.9</td>
<td>152.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>63.3</td>
<td>99.7</td>
<td>429.9</td>
<td>543.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>32.4</td>
<td>37.0</td>
<td>37.9</td>
<td>45.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>64</td>
<td>37.9</td>
<td>45.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>256</td>
<td>67.1</td>
<td>66.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1024</td>
<td>247.7</td>
<td>222.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.27: results for configuration 4 (all times in $\mu$s, $l_{\text{min}} = 8$)

Figure 3.28: results for configuration 4 with Digital MPI (all times in $\mu$s, $l_{\text{min}} = 1$)

118
<table>
<thead>
<tr>
<th>n</th>
<th>n_s</th>
<th>l</th>
<th>t_{r,0}</th>
<th>v_{r,0}</th>
<th>t_{r,100}</th>
<th>v_{r,100}</th>
<th>t_{r,400}</th>
<th>v_{r,400}</th>
<th>t_{r,1600}</th>
<th>v_{r,1600}</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>64</td>
<td>(8.6)</td>
<td>109.9</td>
<td>105.4</td>
<td>9.4</td>
<td>407.3</td>
<td>2.4</td>
<td>1608.2</td>
<td>0.6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>256</td>
<td>(36.1)</td>
<td>38.4</td>
<td>117.4</td>
<td>8.5</td>
<td>414.3</td>
<td>2.4</td>
<td>1620.6</td>
<td>0.6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1024</td>
<td>(95.3)</td>
<td>10.4</td>
<td>165.4</td>
<td>6.0</td>
<td>467.3</td>
<td>2.1</td>
<td>1649.2</td>
<td>0.6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>64</td>
<td>(12.8)</td>
<td>74.0</td>
<td>111.9</td>
<td>8.9</td>
<td>413.8</td>
<td>2.4</td>
<td>1615.2</td>
<td>0.6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>256</td>
<td>(35.3)</td>
<td>27.2</td>
<td>137.1</td>
<td>7.2</td>
<td>432.9</td>
<td>2.3</td>
<td>1639.2</td>
<td>0.6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>1024</td>
<td>(129.6)</td>
<td>7.4</td>
<td>234.1</td>
<td>4.2</td>
<td>535.8</td>
<td>1.8</td>
<td>1737.5</td>
<td>0.5</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>256</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>1024</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

**Figure 3.29:** early results for DAQ61 benchmark suite (all times in μs, frequencies in kHz)

**configuration 4** Results from this configuration serve as a control set and should resemble those from configuration 1, because the fast AlphaStation 500/400, which was not present in the original set of nodes, has been shifted to the least prominent position; there it becomes involved only in few benchmarks. Having such a control set of results is desirable, because the other configurations have been tried at different times. (One node was physically replaced in the meantime.)

**DAQ61 benchmark suite** The server was used as receiver in all runs whose results are quoted in Fig. 3.29. Because of the size of the cluster, the number of senders never exceeded 4. (Fig. 3.29 therefore does not include results for \(n_s = 8\) [9].) The following observations can be made with respect to these results.

- The results for \(t_{r_0}\) in active ROBs and push-farm (3.1) closely resemble those for funnel (1.7) (times in brackets), as was expected. For passive ROBs and pull-farm (3.2) the times in brackets quote accumulated results for both distributing 64 bytes and then collecting 1024-byte messages, as indicated by earlier benchmarks (outfarming (1.5) and funnel (1.7) respectively). Again, these comparisons lead to good matches.

- Performance scales well with the number of senders (“horizontal speedup”), especially if processing times \(d\) are substantial in comparison with communication times.
• Attributing the remaining differences $(t_{r,0} + d) - t_{r,d}$, for $d > 0$, to the effects of overlaps between computation and communication leaves some doubt in the particular case of MEMORY CHANNEL, because in this case minimal observed latencies start already in the same order of magnitude (few $\mu$s) as must also be allowed for inaccuracies of the delay mechanism. MEMORY CHANNEL in general is not a likely candidate for exhibiting such “vertical speedup”, because it adopts a shared memory programming model, where single user-level store instructions suffice to start communication. In the case of absence of a DMA engine these must be performed by the CPU. Other technologies (such as the packet mode of SCI) or other implementations (such as a multi-threaded communication library running on a symmetric multiprocessor node) would offer relative benefits in this regard.

3.3.4 SCI – Scalable Coherent Interface

SCI, the Scalable Coherent Interface [47] — like MEMORY CHANNEL as well — is a contemporary representative of enabling technologies for NUMA architectures. While the second (refer to section 3.3.3) has been adopted by a single vendor for market-ready solutions, SCI is currently more of an open standard, with beginning strong commitments from HP/Convex, Data General⁶, and Sequent⁶. Furthermore, Sun⁷ has recently chosen SCI as interconnect for the Ultra HPC server series, with offerings of currently up to 64 parallel UltraSPARC II processors.

SCI offers more complete services over the present manifestation of MEMORY CHANNEL. In particular, it includes a definition of optional cache coherency (CC-NUMA), and it allows both read and write operations on single mapped virtual addresses.

SCI was devised as a replacement for backplane buses that frequently serve as interconnects in multiprocessor systems (UMA). In order to meet more ambitious design goals, any diverging scenario has to exhibit a clear capacity for high performance and scalability. Although SCI's resulting design shows a paradigm shift from UMA to NUMA (refer to section 3.1), SCI chooses to offer its services as resembling those of an emulated computer bus, whose internal construction from a network of ringlets and switches is effectively hidden from the views of adapters and nodes.

The SCI framework can accommodate up to $2^{16}$ nodes without requiring changes to its specification. Adapters can serve both in full-featured workstations or in passive devices, such as remote memories or peripherals. Its inherent flexibility to use both electrical and optical communication media allows imple-

⁵http://www.dg.com/nmaline/html/sci_interconnect_chipset_and_adapter.html
⁶http://www.sequent.com/numaq/
⁷http://www.sun.com/hpc/tech/interconnect.html
mentations with up to GByte/s of shared bandwidth and kilometers of inter-node distances, in addition to more moderately priced entry-level solutions as well.

The physical-signaling layer of SCI uses fast uni-directional point-to-point links between neighboring nodes. 18 bit wide symbols propagate on the medium at all times. Each symbol includes 2 bytes of traffic data plus one bit for clock and framing information each. Framing serves to assemble unbroken sequences of symbols into discrete packets. Redundant minimally sized packets constitute idle symbols, which are freely inserted to keep the bandwidth utilization constantly at the maximum rate.

SCI’s conceptual node model allows nodes to inject packets into ringlets via their output FIFOs (see Fig. 3.30). Provided that nodes do not make attempts at sending packets that exceed the size of their bypass FIFOs, foreign traffic can be put into temporary hold while sending proceeds. Subsequent idle symbols provide opportunities to reempty bypass FIFOs, after sending has been completed. Packets may only start to originate from an output FIFO when the node’s bypass FIFO is empty.

SCI requires linear sequences of point-to-point segments between nodes to close into circular ringlets. Nodes with connections to more than one ringlet serve as routers, thereby enabling physical representations of more elaborate network topologies, such as grids or cubes. Specialized routers, with otherwise atrophied general functionalities, are offered as SCI switches. Some actual example topologies for a network with up to 5 nodes are shown in Fig. 3.32 on page 125. (The large ring indicates a $4 \times 4$-switch in the drawing.)

One node per SCI ringlet must perform additional housekeeping tasks and act as scrubber. Its function includes the discarding of stalled and orphaned packets after repeated cycles. Orphaned packets cannot be removed by their receivers, as would normally be the case.

The logical layer of SCI utilizes a transaction-oriented protocol between requesters and responders. A transaction typically involves reading or writing data from or to remote memory locations. These locations are identified by 64 bit wide SCI addresses, which are conceptually split into a 16-bit node identifier and a 48-bit offset, for local use within the concerned node. Although the SCI standard leaves the exact interpretation of the lower bits at the discretion of individual responders, they are normally treated as offsets into those pieces of local memory that these nodes make available for shared use over SCI.

This straightforward address decomposition scheme facilitates packet routing, but also leads to non-contiguous addresses for resources that reside on different nodes. This complication, however, is of no direct concern to application-level programming, because programs treat contiguous virtual addresses, which have been converted from SCI addresses via mapping functions, as provided by SCI device drivers.

Node identifiers, command specifiers (for discriminating between protocol steps), sequence numbers, as well as flow control and status information form
SCI packet headers. These are augmented by (optional) data in one of several allowed lengths (staged between 0 and 256 bytes), and a CRC code.

Each transaction consists of up to 2 sub-actions (request and response). Each sub-action in turn involves the exchange of at least 2 packets (send and echo). Their exact number, if higher, depends on the presence of congestion and resulting retransmissions. The move transaction is a form of write operation, which does not include a response sub-action, for greater efficiency in situations where non-acknowledged writes can be tolerated. Fig. 3.30 on the facing page illustrates a generic example for a read (or write) transaction that originates from the upper node.

There are dual relationships between both request/response and send/echo pairs, because the latter tokens always complete loopback cycles that were initiated when a requester sent out the first during earlier protocol steps. Both types of feedback relay different kinds of information.

A positive echo packet acknowledges to the requester that a packet was successfully received in an input queue, and that it therefore need no longer be held in the requester’s output queue for possible retransmissions (local flow control information). A negative echo packet occurs when the original packet had to be discarded, because an input queue temporarily ran out of space. It causes re-attempts at sending.

A response packet on the other hand informs the requester about the completion of the requested operation (or about an error condition). It may return suitable amounts of information, as is obviously appropriate in the case of read transactions.

The lengths of loopback cycles furthermore differ in both cases, if remote transactions between requesters and responders in different SCI ringlets occur. These require active involvement of more nodes as intermediate agents for packet routing. Echo packets originate from the closest intermediate agent (thus they complete only local loops); response packets travel back from the final responder (thus they may complete loops on several ringlets). Intermediate agents volunteer to accommodate routed packets in their output queues during hops across those ringlets, to which they provide direct connections. Intermediate agents act like final responders to requesters, and for final responders an analogous relationship holds. From a node adapter perspective, interfaces to simple ringlets and to complex switched SCI networks are therefore conveniently the same.

We performed measurements on SCI with the following types of hardware: both a PCI-based AlphaServer 4000 5/300 and a PCI-based AlphaStation 500/400 were equipped with Dolphin SCI interface cards, which plugged directly into PCI/PC-slots of these workstations. Two VME-based RIO 8061 embedded processor boards had access to the same types of SCI interfaces via PCI/P PMC-to-PCI/PC converter boards. The technical specifications and performance characteristics of the employed systems were as follows.
Figure 3.30: basic sequence of SCI transactions

Figure 3.31: cache invalidation after SCI transactions
### System Comparison

<table>
<thead>
<tr>
<th></th>
<th>AlphaServer 5/300</th>
<th>AlphaStation 500/400</th>
<th>RIO 8061</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Alpha 21064 EV5</td>
<td>Alpha 2164 EV5.6</td>
<td>PowerPC 604</td>
</tr>
<tr>
<td>clock</td>
<td>299 MHz</td>
<td>400 MHz</td>
<td>100 MHz</td>
</tr>
<tr>
<td>1st level cache</td>
<td>16 kByte</td>
<td>16 kByte</td>
<td>32 kByte</td>
</tr>
<tr>
<td>other caches</td>
<td>96 kByte + 2 MByte</td>
<td>96 kByte + 2 MByte</td>
<td>—</td>
</tr>
<tr>
<td>operating system</td>
<td>Digital UNIX 4.0B</td>
<td>Digital UNIX 4.0B</td>
<td>LynxOS 2.4</td>
</tr>
<tr>
<td>SPECint95</td>
<td>8.11</td>
<td>12.3</td>
<td>n/a</td>
</tr>
<tr>
<td>SPECfp95</td>
<td>12.7</td>
<td>14.1</td>
<td>n/a</td>
</tr>
<tr>
<td>PCI buses</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PCI implementation</td>
<td>64-bit</td>
<td>32-bit</td>
<td>32-bit</td>
</tr>
</tbody>
</table>

The Dolphin SCI card (rev. B) [20] is a 32-bit implementation of the IEEE Standard for SCI [47] that conforms to the PCI local-bus specification. We were able to successfully install the card on the aforementioned systems, on an EB66 embedded processor board (Alpha 21066 with integrated PCI bus interface, under Linux 2.0), and on an industry-standard PC (Pentium 166 MHz, under Linux 2.0). However, it clearly did not tolerate the PCI implementations of some other systems, such as the AlphaStation 200 4/166, for example.

In a typical configuration each SCI card was connected by a station cable (2 m length) to an EDU box (small circles in Fig. 3.32 on the next page), whose in and out connectors attached to link cables that formed the circumference of SCI ringlets. The present implementation used 18-DE-200 link cables for a bit-parallel, electrical implementation of the physical layer, which offered 200 MByte/s aggregate bandwidth on the medium. Minimum configurations consisted of 2 systems, whose SCI cards were directly connected via station cables (thus effectively forming ringlets of 2 nodes). We also used a 4 × 4 switch (large circles in Fig. 3.32) for building structured configurations. Link cables of ≤ 10 m length were available and in use, and the connected systems were physically distributed over several rooms.

#### 3.3.4.1 Software configuration

The left part of Fig. 3.34 on page 126 shows the protocol stack that was used for enabling message passing over SCI. The purposes of individual protocol layers that appear in the stack are as follows.

**SCI PHY-API** The SCI PHY-API is the subject of an ongoing standardization effort that will finally lead to a new IEEE standard [48], which offers rules on how to grant low-level software access to SCI networking services in a device-independent way. The SCI PHY-API is concerned with controlling the mapping between SCI addresses and virtual addresses (API addresses) and is meant as a low-level abstraction from SCI hardware, for use by application programs.

The scope of SCI addresses encompasses the entire cluster; thus SCI addresses are a convenient means for uniquely identifying cluster-wide memory resources.
Figure 3.32: cluster configurations for SCI measurements

<table>
<thead>
<tr>
<th>SCI PHY-API</th>
<th>POSIX P1003.1b</th>
<th>SVR4</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCIInitialize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCIMkSharedMemory</td>
<td>shm_open</td>
<td>shmat</td>
</tr>
<tr>
<td>SCIMapAPIWindow</td>
<td>mmap</td>
<td></td>
</tr>
<tr>
<td>SCIMapSCIWindow</td>
<td>munmap</td>
<td>shmat</td>
</tr>
<tr>
<td>SCIUnMapAPIWindow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCIUnMapSCIWindow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCIFreeSharedMemory</td>
<td>shm_unlink</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.33: related standards governing shared memory access

125
Figure 3.34: Protocol stacks for ATLAS communication benchmarks

Figure 3.35: Overview of cluster-wide shared data structures
Virtual addresses may be reused on different nodes with different local semantics; they are vital, because all executable code is expressed in terms of virtual addresses.

An example implementation of the SCI PHY-API (draft 0.41)\(^8\) for Dolphin SCI cards [34] has become recently available at CERN. It succeeds in offering an abstract view of Dolphin's chip-set, which no longer requires knowledge of individual registers, and enables a subset of the currently specified semantics.

We used it in a mode where, relative to each mapping between nodes, one node acted as server and the others acted as its clients. The server was the node in whose physical memory the shared data actually resided. (Only if optional cache coherency is supported, copies may reside in caches on the clients' sides as well.) Without cache coherency it is more straightforward to implement message passing such that the receiver coincides with a server, for otherwise multiple accesses to received data lead to repeated transports across the SCI network.

The following steps lead to connection establishment between 2 nodes, if executed in precise order in a distributed environment (see Fig. 3.36; refer also to upper part of Fig. 3.35 on the preceding page): the server reserves a portion of mappable physical memory, which later holds shared data for both participating nodes (SCIMkSharedMemory). It converts the portion's local virtual address into an SCI address, which it can advertise among possible clients (SCIMapSCIWindow). Clients receive this information and convert the SCI addresses back into virtual addresses (SCIMapAPIWindow), for use by locally executing processes. Once processes on both nodes have determined local virtual addresses that relate to corresponding global SCI addresses the mapping is established and communication may proceed.

**Related Standards** The SCI PHY-API is at the core of the SCI-specific protocol stack that is shown in Fig. 3.34 on the facing page. Its emphasis is on offering a vendor-independent abstract view of the SCI hardware for the benefit of software that resides on upper layers of the protocol stack. The ATLAS communication benchmarks require that the programming model changes from shared memory access to message passing in upper layers of the protocol stack. The data structures that are used for accomplishing this transition are shown in Fig. 3.35 on the preceding page.

Alternatively, a similar effort could aim at bridging the semantic gap between the SCI PHY-API and abstractions from other APIs that are in widespread use for accessing shared memories. This horizontal extensions at the level of the SCI PHY-API would still adhere to the shared memory paradigm and would help to increase the level of abstractness from vendor-independence towards even technology-independence. In particular, it would completely hide the differences between centralized and distributed shared memories from applications' views,
although with the present generation of Dolphin SCI cards (cache coherency not yet supported) it is not clear whether this is already practical or desirable.

The rows in Fig. 3.33 on page 125 show the approximate semantic relationship between calls from SCI PHY-API, POSIX real-time extensions [46], and UNIX System V Release 4 [27]. Different background shades indicate whether functions are of concern to servers and/or clients, according to the previous description.

**Assumptions made on higher level-software** This implementation requires that a local copy of the daemon scid runs on each node that takes part in communication over SCI. (The daemon comes as part of the example implementation of the SCI PHY-API.) It addresses some aspects of cluster configuration management, such as name resolution. A more mature implementation of the daemon could alternatively combine all of the following responsibilities.

- Processes that use the SCI PHY-API as clients require knowledge of SCI addresses that are offered by remote servers before connection establishment. Methods for exchanging SCI addresses between nodes are outside the domain of the proposed standard. SCI addresses must be dynamically exchanged, because they may vary between runs (subject to auto-configuration rules and at the discretion of SCI drivers).

While different possible solutions can be envisaged from a technical point
of view (configuration-file in commonly mounted file-system, usage of CSR-space registers with cluster-wide visibility), we currently favor a scheme which uses TCP/IP sockets for exchanging relevant setup information between senders and receivers.

**scid** keeps information regarding mappings in an in-memory database, and allows remote queries over TCP/IP (port 8192). This solution can be ported with little effort to a wide range of platforms, provided that all nodes that participate in SCI communication have access to TCP/IP services (usually over Ethernet, potentially over SCI as well). The daemon’s client-interface includes the following functions, among others: **sci_map** enters node ids, memory ids, API addresses, window sizes, and process ids into the local database. **sci_lookup** retrieves this information from remote nodes by their IP host names.

The daemon’s overall functionality is, at present, strictly limited to the handling of this database. It does for instance not accept and forward SCI-related requests from remote to local processes, although with the provision of a suitable IPC protocol it could easily accomplish this task. In such a scenario a single server per cluster could allocate all distributed SCI resources at startup-time via remote IPC calls, whereas currently typically several servers have to become active per run. (This and other daemon functions could alternatively also be enabled on the level of local SCI drivers, as it is the case for instance with Digital TruCluster software for Memory Channel.)

- The daemon in turn requires knowledge of remote host names in the cluster. As a simple method for providing them, this implementation assumes that the local environment variables **SCINODEx** \(x \geq 0\) are set to these proper host names. The implicit order of host names is also used to determine unique process ranks (corresponding to the index \(x\) of the environment variable **SCINODEx** that matches a given host name). The current implementation depends on equal settings of these environment variables on all cluster nodes.

Alternatively, the provision of a single host name would suffice under an extended scheme, where nodes with distinct host names would have to register with the selected node over TCP/IP. The selected node would then obtain and redistribute a dictionary of relevant host names within the cluster. Yet another solution could use initial IP broadcasting, provided that all nodes reside on the same IP subnet.

Such asymmetric configurations could lead to implementations where only a single daemon remains, and where other peers’ rudimentary tasks are merged with the client interfaces that is linked into applications. (The
functionality of the client interface could in turn migrate into the SCI PHY-API.)

- In order for SCI to work properly, the interface of exactly one node per ring must be selected as scrubber. (Scrubbers are indicated by hexadecimal labels that appear in white prints on black backgrounds in Fig. 3.32 on page 125.)

Although the SCI standard suggests that scrubbers should be selected during auto-configuration by a scheme that is transparent to application programs, this functionality is not yet supported in the current generation of Dolphin’s SCI cards. Therefore scrubbers have to be chosen manually after each (re)initialization of interfaces. (Jumper-settings can be used alternatively as well, but with frequently changing configurations this provides even less convenience.)

A more sophisticated server could deal with scrubber selection and thus reduce latent errors that are due to manual intervention, although it would then have to obtain knowledge of the SCI network’s topology (number of scrubbers) in addition as well.

- Daemons could exchange occasional probe messages over SCI, in order to confirm that the physical network connections are still uncompromised and that connections remain in state “alive”. This would offer valuable aid in laboratory environments (especially to remote users), where both cabling and low level communication software are frequently touched and updated.

- Alternatively, the daemon could accept connections on a further TCP/IP port. This could be used for status reporting to (for instance) Java-based surveillance programs, which would typically display their results as hypertext on WWW.

**SCI driver** The portable SCI driver currently only offers some minimal functionality that is related to mapping CSR-space registers into user space, where they can be accessed and manipulated by application programs. (At present, there is in fact a restriction to only one client per driver at each time.)

Much functionality that should optimally reside on the driver level (e.g. for security reasons) is still implemented in user-level libraries. The reason for having adopted this temporary solution first was that driver (re)development involves considerably longer turn-around times and exposes target systems to larger risks of global failure in case of initial (driver) software or device malfunctions.

It is envisaged that some functionality will finally migrate from the user level (back) into a proper device driver. Suggestions as to the extent of such shifts have been made by Dolphin (they already partly appear in the company’s offerings
for SBus-based systems and PCs) and others, although we suggest that at this time the exact definition still needs e.g. better understanding of the conventions that are differently adopted by the UNIX, Window NT, and Lynx OS operating systems. (ioctl(2) and related system calls are largely UNIX-dependent and also don’t map well onto connection-less services.)

The driver’s extension — although clearly desirable from a software engineering point of view — is currently assigned only low relative priority, because its changes will not have any major effects on communication performance. We foresee that the further development of drivers and related software will have to be pursued mainly by industry, for design decisions relating to the ATLAS IVL2 trigger will partly depend on whether full commercial hardware and software support will become available for candidate technologies in due time.

### 3.3.4.2 Implementation of the low level interface

The software for the implementation of the low level interface was structured such that a single directory tree can support several target systems. The make-files put libraries and binaries automatically into subdirectories whose names are constructed such as to indicate the types of supported CPUs (uname -m) and operating systems (uname -s) (e.g. PowerPC-LynxOS/). Debugging modes for printing verbose diagnostics, for transmitting random test messages, and for controlling checksums are optionally available at compile-time (refer to section 3.3.3).

The current implementation assumes that the SCI PHY-API offers access to SCI transparent mode. Therefore it does not use transparent transactions (such as SCIWrByte), but rather includes the corresponding assignment statements directly in the code.

Communication occurs in \( n \times n \) buffers in SCI address space (shown in matrix arrangement in Fig. 3.35 on page 126; see also Fig. 3.18 on page 112), where \( n \) is the total number of communicating processes. Every node serves one column of buffers (drawn with white backgrounds) and is client relative to the others (drawn with dark backgrounds). The matrix is indexed horizontally by the ranks of receiving processes, and vertically by the ranks of sending processes.

Message data arrive in buffers that are physically located at receivers’ sides (at the tips of thick arrows in Fig. 3.35). A buffer’s contents are valid (still required) if any of 2 associated flags is set. The protocol requires that both flags are always updated in sync, therefore senders and receivers can choose which flag they want to inspect at their conveniences. One flag resides with the buffer on the receiver’s side (local_flag), the other is located in physical memory on the sender’s side (remote_flag, diagonally across the matrix in Fig. 3.35).

The following protocol is used for handshaking between senders and receivers: senders wait until a buffer becomes available for (re)use by repetitively checking whether it is still guarded by a flag that is set. In order not to have to poll the flag’s value across the network (without support of cache-coherent SCI this...
would cause 2 SCI sub-actions per loop cycle) the sender inquires remote_flag. When the polling succeeds, the sender copies the message data into the receiver’s memory and then sets both flags. On the other end receivers poll local_flag for detecting when a new message has arrived. They then extract the message data into buffers that are privately held by applications and reset both flags.

The implemented version includes the following optimizations and deliberate restrictions in its design.

- There are no queues of buffers, but only single buffers, held between senders and receivers. Queues would allow senders to maintain non-blocking communication semantics even if receivers were continually lagging behind in operation. This is helpful when the relative rates at which senders and receivers call communication primitives fluctuate due to the myriad effects of intermediate calculations. If senders are permanently faster than receivers, queues can only make a difference during a short start-up phase, while they are in the process of filling up.

In contrast, the little amount of calculation that is included in the ATLAS communication benchmarks does (almost) not vary in duration between transfers. Results from a similar MEMORY CHANNEL-related implementation (refer to section 3.3.3) indicate that queues even impose a small communication penalty in the particular case of these benchmarks. This is perhaps related to the fact that data and control information become dispersed over several cache lines (or pages) as data structures grow in size (see Fig. 3.19(a)), thus causing more net traffic.

- The implementation avoids repetitive retrievals of shared data (e.g. polling of flags) across the SCI network. Therefore it tolerates non-cache-coherent SCI implementations without performance losses.

- The implementation is prepared to allow for write gathering as a means of performance optimization, by making proper use of SCIBarrier calls in conjunction with the manipulation of flags. While this involves writing into a CSR-space register, which has been reported as generally slow, it is perhaps preferable to relying on timeouts for forcing SCI packets out, even if address counts have not yet reached 64-byte boundaries. (Only of concern if write gathering is on.)

- Both the Alpha’s and PowerPC’s superscalar architectures allow weakly ordered memory accesses. Thus these CPUs’ load and store instructions do not have to complete in the order in which they appear in the program text. This feature provides opportunities for improving dynamic arbitration of CPU instructions among execution units and for avoiding processor stalls at the chip-design level. As a trade-off, the remaining dependencies on a particular order must be marked by the inclusion of mb (memory barrier)
or \texttt{eieio} (enforce in-order execution of I/O) instructions, depending on the
type of processor. These instructions place barriers in the flow of mem-
ory access operations. Their special semantics is such that all previously
initiated instructions appear to have completed before, and all subsequent
instructions appear to be initiated only after their completion.

The code that implements the low level interface uses these barrier instruc-
tions before setting (or resetting) flags, in order to prevent confirmations
of partially transmitted messages (or corruption of buffers by early arriving
follow-up messages).

- Alpha and PowerPC CPUs use different byte-orderings by default (little-
endian and big-endian respectively). The communication library transmits
its control information in Internet network byte order, but does not convert
message data. This task is left to application programs, partly because full
type information is required for performing any conversions. Contrary to
MPI's API, parameterized type information is not passed during calls via
the low level interface.

3.3.4.3 Results for ATLAS communication benchmarks

Individual mini-applications from the ATLAS communication benchmarks suite
are referred to by the numbering scheme that was introduced in section 3.2.

The software that implements the protocol stack currently compiles under
Digital UNIX, LynxOS, and Linux (version for Alpha CPUs). The parts above
the driver level also compile under Windows NT and Linux (version for Intel PCs).
A driver for the last 2 platforms is currently available from Dolphin. It could be
used for enabling the ATLAS communication benchmarks, if the SCI PHY-API
implementation were changed to conform to its own interface.

Results for all measurements are given in Figures 3.40 to 3.42. (The left
graphs in Fig. 3.37 to Fig. 3.39 as well as in Fig. 3.43 plot packet sizes in bytes
vs. observed durations in $\mu$s; the right graphs plot packet sizes in bytes vs. av-
erage bandwidths in MByte/s.) Individual benchmarks executed with repetition
counts of $c \geq 10000$. Compiler optimization was turned on for all architectures.
(We used DEC C V5.2-033 and GNU C 2.6-95q2 for Alphas and PowerPCs re-
spectively.) Message copying was done with \texttt{memcpy(3)} on the AlphaServer and
AlphaStation (this turned out to be most efficient for $l \geq 16$ kByte, in partic-
ular), and with an optimized loop encompassing values of type \texttt{long long} (a
GNU C-specific extension for accessing 64-bit quadwords) on the RIOs.

All times were quoted from \texttt{clock(3)}. The large repetition counts are meant
to cover for the coarse granularity of this timer ($\gg 100 \mu$s). Because of SCI's ad-
hherence to the shared memory programming model, communication is driven by
code that resides entirely on the user level (the SCI driver's kernel-level services
are accessed only at initialization time). The code does not perform any system
calls while the clock is running (the application does not voluntarily yield control to other processes). Thus we prefer \texttt{clock}(3)’s measurement of CPU-time as sufficiently accurate and more relevant over wall-clock time (as reported by \texttt{gettimeofday}(2)) in the particular case of SCI. (This would not generally hold for ATM for instance, because it misses out on any CPU consumption during driver activations.)

In the course of these measurements we changed between several configurations and parameter settings as follows, in order to show their relative mutual effects.

**Different programming models and optimization levels** The vertical decomposition of Fig. 3.40 on page 136 refers to the use of different optimization levels for communication hardware and software, as follows. Best results appear close to the top.

- **upper aspect:** *Write gathering* is a feature that can optionally be enabled for Dolphin SCI cards. It instructs the interface to preferably send larger packets, by delaying outgoing traffic until 64-byte address boundaries are crossed during writes into mapped memory. 

  \texttt{DONT TOUCH BUFFER} is an option whose effect is limited to the ATLAS communication benchmarks. In order to provoke cache misses and, more generally, prevent “clever” run-time environments from actually moving data that is not processed by receivers, these programs include code that “touches” message buffers on both the sender’s and receiver’s sides, by default. Touching consists of loops that access all bytes in a message individually. Especially when Alpha and SCI technologies were combined, they caused considerable extra overheads, as were not seen with earlier measurements on other systems. The compile-time macro \texttt{DONT TOUCH BUFFER} disables the corresponding code and thus factorizes the effect out.

A combination of both write gathering and \texttt{DONT TOUCH BUFFER} allowed us to obtain best communication results over SCI for the particular low level interface (two-copy implementation). Latencies for small messages (8 bytes) were \( \sim 10 \, \mu \text{s} \) (calculated from one-way (1.1) as \( t_s/2 \)), and asymptotic bandwidths were between \( \sim 15 \, \text{MByte/s} \) (between RIOs 8061 in configuration 3) and \( \sim 52 \, \text{MByte/s} \) (between AlphaStation 500/400 and AlphaServer 4000 5/300 in configuration 2), both asymptotic bandwidths calculated from pairs (1.4) as \( l/T^C \)). Asymptotic bandwidths were reached starting from packet sizes of \( l \approx 4 - 8 \, \text{KB} \). One-copy implementations can offer considerably better performance in comparison, in particular in communication between Alpha-based systems (see Fig. 3.37 and below).

- **middle aspect:** These results are quoted for allowing direct comparisons with older sets of measurements for other hardware platforms. (The macro
Figure 3.37: results from AlphaStation 500/400 to AlphaServer 4000 5/300

Figure 3.38: results from RIO 8061 to AlphaServer 4000 5/300

Figure 3.39: results from RIO 8061 to RIO 8061

135
<table>
<thead>
<tr>
<th>$n$</th>
<th>$l$</th>
<th>$t_s$</th>
<th>$\bar{t}$</th>
<th>$\bar{t}_r$</th>
<th>$t_s$</th>
<th>$\bar{t}$</th>
<th>$\bar{t}_r$</th>
<th>$t_s$</th>
<th>$\bar{t}$</th>
<th>$\bar{t}_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>25.0</td>
<td>18.1</td>
<td>9.9</td>
<td>19.9</td>
<td>14.9</td>
<td>8.3</td>
<td>20.0</td>
<td>16.5</td>
<td>9.0</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>25.0</td>
<td>16.3</td>
<td>13.3</td>
<td>18.3</td>
<td>13.3</td>
<td>9.9</td>
<td>22.0</td>
<td>14.5</td>
<td>13.0</td>
</tr>
<tr>
<td>2</td>
<td>256</td>
<td>36.0</td>
<td>33.1</td>
<td>19.9</td>
<td>24.9</td>
<td>14.9</td>
<td>13.3</td>
<td>41.0</td>
<td>36.5</td>
<td>23.0</td>
</tr>
<tr>
<td>2</td>
<td>1024</td>
<td>99.0</td>
<td>98.9</td>
<td>48.3</td>
<td>38.3</td>
<td>52.4</td>
<td>26.6</td>
<td>125.0</td>
<td>106.5</td>
<td>62.0</td>
</tr>
<tr>
<td>2</td>
<td>2048</td>
<td>172.4</td>
<td>179.9</td>
<td>86.6</td>
<td>98.3</td>
<td>91.6</td>
<td>46.6</td>
<td>227.0</td>
<td>196.0</td>
<td>112.9</td>
</tr>
<tr>
<td>2</td>
<td>4096</td>
<td>323.0</td>
<td>340.3</td>
<td>161.6</td>
<td>174.9</td>
<td>166.6</td>
<td>84.9</td>
<td>435.0</td>
<td>382.0</td>
<td>216.9</td>
</tr>
<tr>
<td>2</td>
<td>8192</td>
<td>834.0</td>
<td>803.4</td>
<td>309.9</td>
<td>333.3</td>
<td>318.3</td>
<td>161.6</td>
<td>1259.0</td>
<td>911.0</td>
<td>435.6</td>
</tr>
<tr>
<td>2</td>
<td>16384</td>
<td>1606.0</td>
<td>1642.1</td>
<td>639.9</td>
<td>639.9</td>
<td>611.6</td>
<td>313.3</td>
<td>2461.5</td>
<td>2028.0</td>
<td>1179.2</td>
</tr>
</tbody>
</table>

Figure 3.40: results for message passing between different types of nodes (all times in $\mu$s)

<table>
<thead>
<tr>
<th>$n$</th>
<th>$l$</th>
<th>$t_s$</th>
<th>$\bar{t}$</th>
<th>$\bar{t}_r$</th>
<th>$t_s$</th>
<th>$\bar{t}$</th>
<th>$\bar{t}_r$</th>
<th>$t_s$</th>
<th>$\bar{t}$</th>
<th>$\bar{t}_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>32.0</td>
<td>21.8</td>
<td>13.3</td>
<td>23.3</td>
<td>18.3</td>
<td>9.9</td>
<td>24.0</td>
<td>19.0</td>
<td>11.0</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>60.0</td>
<td>46.3</td>
<td>41.6</td>
<td>28.9</td>
<td>21.6</td>
<td>16.6</td>
<td>61.0</td>
<td>35.5</td>
<td>32.0</td>
</tr>
<tr>
<td>2</td>
<td>256</td>
<td>195.0</td>
<td>149.9</td>
<td>143.3</td>
<td>71.9</td>
<td>43.3</td>
<td>34.9</td>
<td>208.0</td>
<td>113.0</td>
<td>105.0</td>
</tr>
<tr>
<td>2</td>
<td>1024</td>
<td>746.0</td>
<td>590.0</td>
<td>549.0</td>
<td>245.9</td>
<td>134.1</td>
<td>121.6</td>
<td>799.0</td>
<td>436.0</td>
<td>398.0</td>
</tr>
<tr>
<td>2</td>
<td>2048</td>
<td>1472.0</td>
<td>1167.3</td>
<td>1089.9</td>
<td>476.6</td>
<td>256.6</td>
<td>238.3</td>
<td>1583.0</td>
<td>857.0</td>
<td>790.0</td>
</tr>
<tr>
<td>2</td>
<td>4096</td>
<td>2921.0</td>
<td>2323.2</td>
<td>2173.2</td>
<td>950.6</td>
<td>501.6</td>
<td>464.9</td>
<td>3161.0</td>
<td>1705.5</td>
<td>1572.0</td>
</tr>
<tr>
<td>2</td>
<td>8192</td>
<td>5569.0</td>
<td>4786.4</td>
<td>4338.1</td>
<td>1901.5</td>
<td>1049.9</td>
<td>924.9</td>
<td>6611.0</td>
<td>3552.0</td>
<td>3161.0</td>
</tr>
<tr>
<td>2</td>
<td>16384</td>
<td>12247.0</td>
<td>9522.4</td>
<td>8727.9</td>
<td>3744.5</td>
<td>2200.2</td>
<td>1864.9</td>
<td>13178.0</td>
<td>6988.5</td>
<td>6618.0</td>
</tr>
</tbody>
</table>

Figure 3.41: results for message passing in different topologies (all times in $\mu$s)
configuration 1

3.1

<p>| $\sigma_p^{\text{just}}$ ± $\sigma_p^{\text{stat}}$ | $\pm 0.9 \pm 0.8$ | $\pm 4.2 \pm 0.8$ | $\pm 17.5 \pm 0.8$ |</p>
<table>
<thead>
<tr>
<th>$n$</th>
<th>$n_s$</th>
<th>$n_r$</th>
<th>$l$</th>
<th>$t_{r,0}$</th>
<th>$v_{r,0}$</th>
<th>$t_{r,100}$</th>
<th>$v_{r,100}$</th>
<th>$t_{r,400}$</th>
<th>$v_{r,400}$</th>
<th>$t_{r,1600}$</th>
<th>$v_{r,1600}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>64</td>
<td>14.9</td>
<td>(13.3)</td>
<td>67.1</td>
<td>104.2</td>
<td>9.5</td>
<td>407.5</td>
<td>2.4</td>
<td>1610.7</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>256</td>
<td>19.9</td>
<td>(19.9)</td>
<td>50.2</td>
<td>105.8</td>
<td>9.4</td>
<td>405.8</td>
<td>2.4</td>
<td>1609.1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1024</td>
<td>49.9</td>
<td>(48.3)</td>
<td>20.0</td>
<td>107.5</td>
<td>9.3</td>
<td>409.1</td>
<td>2.4</td>
<td>1615.7</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>64</td>
<td>14.9</td>
<td>(14.9)</td>
<td>67.1</td>
<td>107.5</td>
<td>9.3</td>
<td>407.5</td>
<td>2.4</td>
<td>1607.4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>256</td>
<td>23.3</td>
<td>(23.3)</td>
<td>42.9</td>
<td>107.5</td>
<td>9.3</td>
<td>410.8</td>
<td>2.4</td>
<td>1610.7</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1024</td>
<td>49.9</td>
<td>(49.9)</td>
<td>20.0</td>
<td>117.5</td>
<td>8.5</td>
<td>417.5</td>
<td>2.3</td>
<td>1622.4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>64</td>
<td>21.6</td>
<td>(21.6)</td>
<td>46.2</td>
<td>109.2</td>
<td>9.1</td>
<td>410.8</td>
<td>2.4</td>
<td>1609.1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>256</td>
<td>24.9</td>
<td>(23.3)</td>
<td>40.1</td>
<td>112.5</td>
<td>8.8</td>
<td>414.1</td>
<td>2.4</td>
<td>1615.7</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1024</td>
<td>76.6</td>
<td>(76.6)</td>
<td>13.0</td>
<td>124.2</td>
<td>8.0</td>
<td>425.8</td>
<td>2.3</td>
<td>1629.1</td>
</tr>
</tbody>
</table>

write gathering, DONT TOUCH BUFFER

3.2

<table>
<thead>
<tr>
<th>$t_{r,0}$</th>
<th>$v_{r,0}$</th>
<th>$t_{r,100}$</th>
<th>$v_{r,100}$</th>
<th>$t_{r,400}$</th>
<th>$v_{r,400}$</th>
<th>$t_{r,1600}$</th>
<th>$v_{r,1600}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>71.6</td>
<td>(61.6)</td>
<td>13.9</td>
<td>165.8</td>
<td>6.0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>88.3</td>
<td>(71.5)</td>
<td>11.3</td>
<td>167.5</td>
<td>5.9</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>98.3</td>
<td>(109.9)</td>
<td>10.1</td>
<td>202.5</td>
<td>4.9</td>
</tr>
</tbody>
</table>

Figure 3.42: results for DAQ61 benchmark suite (all times in $\mu$s, frequencies in kHz)

Figure 3.43: results for distributed vs. centralized shared memory systems

137
DONT TOUCH BUFFER is always turned off by default.)

As can be seen from Figures 3.37 to 3.40, this leads to an important decline in observed bandwidth by $\sim 20$ MByte/s for communication between fast Alpha CPUs. The decline is less prominent for communication between a RIO 8061 and AlphaServer 4000 5/300 and it is practically absent for communication between RIOs 8061. It is perhaps related to the lack of byte manipulation instructions (and therefore need for emulation) in Alpha CPUs prior to revision EV5.6.

- lower aspect: Results in this part clearly show that write gathering has a very strong effect on optimization and must be considered a mandatory option. Communication software must be prepared for its usage either by aligning all data structures to 64-byte boundaries or by making proper use of SCIBarrier calls, if one does not wish to rely on the (expensive) time-out mechanism for flushing partially filled 64-byte blocks. (As an effect of write gathering, SCI performs even better on 64-byte messages than on smaller sizes, if the feature is enabled.)

- pseudo-one-copy implementation: The lines that correspond to the highest bandwidths in Figures 3.37 to 3.39 were obtained by temporarily disabling (commenting out) the second copy in the implementation of the two-copy message passing library. We do not include precise numbers in Fig. 3.40, because this crudely optimized version was of course unable to deliver data correctly. (Still, many of the ATLAS communication benchmarks were not affected, thanks to their minimal computational semantics.) It allowed us to arrive at estimates on how fast a one-copy implementation can operate at best, because the flow control mechanism for granting buffer access between senders and receivers via handshake still remained in place; i.e., control messages passed back and forth.

Actual bandwidths observed with a one-copy communication library of the type outlined in section 3.2.2 would typically be lower, because disabling the second copy implicitly carries the assumption that either an unlimited amount of buffer space is available on receivers’ sides, or receivers operate permanently at a faster rate than senders. Therefore present estimates on the effect of two-copy vs. one-copy implementations use a conservative lower bound (two-copy implementation without queues; some overhead caused by mini-applications) and an optimistic upper bound (infinite buffering resources or slow senders). We expect the actual margin to be narrower.

The observed difference for two-copy vs. one-copy implementations is most prominent for communication between Alpha-based systems, where results indicate gains in asymptotic bandwidth of $\sim 70$ MByte/s over $\sim 52$ MByte/s (configuration 2). The upper margin is well in line with other reported measurements concerning Dolphin SCI cards [35].
Different types of nodes The results from Fig. 3.40 on page 136 indicate how the involvement of different types of nodes (different computing speeds, different PCI implementations) effects communication performance. Configuration 1 (see Fig. 3.32 on page 125) highlights communication between a RIO and the AlphaServer 4000 5/300. This shows medium fast performance in a quantitative comparison (the less sophisticated PCI implementation on the RIOs and their lack of a level-2 cache perhaps constitute the most prominent degrading factors), but is closest to the currently envisaged requirements of the ATLAS LVL2 trigger, where RIOs are foreseen as candidate data sources (ROBs). Configuration 2 enables communication between the AlphaStation 500/400 and AlphaServer 4000 5/300, i.e. between the fastest types of available machines. Communication performance is clearly superior to what was measured between 2 RIOs in configuration 3, which is also well in line with obvious expectations.

The bends in Fig. 3.39 on page 135 toward lower bandwidths for $l > 8$ kByte are perhaps related to internal caching of the PowerPC chip (16 kByte primary data cache). The bend is absent from Fig. 3.38 on page 135, because in this case the RIO 8061 acts only as sender, not as receiver. The sender’s memory accesses are mostly into mapped memory and are therefore less strongly influenced by cache sizes.

While communication was stably maintained between the AlphaServer 4000 5/300 and AlphaStation 500/400 during numerous runs, with repetitions counts of $c \geq 10000$, we observed reoccurring problems involving the RIOs 8061. Several runs had to be repeated 2 or 3 times in succession, in order to make all messages successfully pass across the network. It is currently unknown whether this malfunction must be attributed to software or hardware problems. Software problems relating to NFS operation have been known to exist on the RIOs 8061 already for some time. The current mechanical setup of PCI cards that attach to the RIOs’ VME boards and their cooling in the VME crate is hardly satisfactory and may also constitute an important negative influence. The latter problem will disappear when the next generation of Dolphin SCI cards (in PCI/PMC format) will become available.

Different Topologies Fig. 3.41 on page 136 shows the effects of different network topologies on communication performance. (Refer to Fig. 3.32 on page 125 for a graphical explanation of these topologies.) The results indicate that every extra node per ringlet causes a latency increment in the order of $0.5 - 1 \mu s$. Performance over the tested $4 \times 4$ switch suggests that ringlets of more than $\sim 4 - 5$ nodes should be broken up into several smaller arrangements and connected via switches, if hardware acquisition costs permit.

Representatives of UMA vs. NUMA Even with these encouraging performance numbers, it is important to keep in mind that centralized shared memory
systems (UMA) outperform distributed shared memory systems (NUMA) as long as their central memory access paths are not becoming sparse resources. This perhaps partly explains why Digital currently offers MEMORY CHANNEL only as a fast communication medium between AlphaServers, which typically come in multiprocessor configurations (up to 12 CPUs), and which are built around centralized shared memories. This observation holds, although the adapters and software would very well work with smaller AlphaStations as well (refer to section 3.3.3).

On the other hand, SCI (as another example of an enabling technology for distributed shared memories) has the following (partly specific) advantages over centralized shared memory architectures.

- It permits the shared memory programming model to scale smoothly to configurations with more than 4 (Intel Pentium Pro) to twelve (Digital Alpha) CPUs.

- It it is able to accomplish the transport of data into shared memories and across physical distances in a way that is consistent with continuing use the same programming model. (This is of important concern to intended applications in the domain of the ATLAS LVL2 trigger.)

- Heterogeneous computing clusters of potentially low cost might become a realistic option, because the adapters can operate in conjunction with a variety of PCI-based systems from different vendors (industry-standard PCs, in particular) and operating systems (Linux and Windows NT, in particular).

Fig. 3.43 on page 137 quantifies the difference between observed latencies and bandwidths. We chose the following systems to represent NUMA and UMA designs (in line with given availabilities).

Results for distributed shared memory (NUMA) were quoted from what has been measured with the one-copy version of the communication library for SCI (optimistic estimates, see above) between an AlphaStation 500/400 and AlphaServer 4000 5/300.

Results for centralized shared memory (UMA) were obtained on an AlphaServer 8400 5/440 (10 CPUs, Alpha 21164 at 437 MHz, 96 kByte + 4 MByte cache per CPU, 4 GByte RAM). The peak in the bandwidth plot at 8 kByte is perhaps caused by the size of the chips’ primary caches (8 kByte Dcache). The plotted results constitute a conservative estimate, because UMA permits even zero-copy communication libraries.

The 2 configurations were reasonably similar to each other apart from the employed communication technology, whose effect we wanted to study. Each CPU on the high-end AlphaServer 8400 5/440 outperformed similar CPUs on the other 2 systems due to higher clock rates and larger private caches. On the
other hand, the large system was in production use while the benchmarks were run, with \( \sim 4 \) CPUs running individual jobs with \( > 90\% \) CPU loads.

**DAQ61 benchmark suite** The results in Fig 3.42 on page 137 indicate that an AlphaServer 4000 5/300 can sustain reception of 1-kByte messages from 3 RIOs 8061, in parallel, at a rate of \( \sim 13 \) kHz, if the two-copy implementation of the message passing library for SCI is employed. The rate increases to \( \sim 20 \) kHz if there are only 2 senders. It drops to \( \sim 10 \) kHz and \( \sim 11 \) kHz respectively for a pull-oriented (instead of push-oriented) scheme.

In a full trigger system many sets of senders and receivers would operate in parallel, thus it would not be necessary for a single funnel to reach 100 kHz (the required overall frequency for the ATLAS LVL2 trigger).

Adding more receivers (LVL2 processors) would help, but only to the extent to which the SCI network could handle the combined load. (The present version of the benchmarks allows only studies with one receiver each.) Results from Fig 3.40 on page 136 indicate that a single RIO 8061 can send 1-kByte messages to an AlphaServer in 26.6 \( \mu s \) (at \( \sim 37 \) kHz); i.e., the present limitation is not only imposed by the receiver (the obvious bottleneck in the funnel) alone.

Measurements that are discussed in this section were performed during normal daytime operation of the system. The AlphaServer 4000 5/300 handled interactive logins from several other users who ran shells or editors (\( < 1\% \) CPU load), and exported file systems to about a dozen (mostly idle) NFS clients.

### 3.3.4.4 Conclusions

- The presented benchmarks use a message passing library that resembles a subset of MPI and that makes use of SCI transparent mode via the SCI PHY-API. The library’s design was deliberately kept simple for pragmatic reasons. (Its implementation and the gathering of results took only few weeks.) Therefore it copies all data twice during transport, and it does not provide queues for buffering between senders and receivers.

- All measurements were obtained with `clock(3)` (not `gettimeofday(2)`).

- Unidirectional ping-pong latencies for small messages were \( \sim 10 \mu s \). Asymptotic bandwidths were measured at \( \sim 52 \) MByte/s between Alpha-based workstations (two-copy implementation); they occurred starting from message sizes of \( l \approx 4 - 8 \) kByte.

- With an experimental pseudo-one-copy version of the communication library bandwidths over SCI were already close to what PCI can realistically deliver by today. The relative discrepancies (\( \sim 70 \) MByte/s between Alpha-based workstations, and \( \sim 30 \) MByte/s when RIOs 8061 act as senders)
suggest a strong correlation between SCI performance and the quality of particular PCI implementations.

- An AlphaServer 4000 5/300 (which emulated a LVL2 processor for ATLAS) can receive 1-kByte messages over SCI in push-mode and in parallel from 3 RIOs 8061 (ROBs for ATLAS) at a maximum sustained rate of $\sim 13$ kHz. In pull-mode (64-byte request messages) this rate decreases to $\sim 10$ kHz (see Fig. 3.42 on page 137). These results can be improved on by specifically optimizing generic code towards SCI. The observed times indicate that the application-specific communication overhead (the part that cannot be overlaid with computation if only a single CPU and no coprocessor is present) on the receiver’s side is $\sim 10 - 30$ µs.

- For each reported measurement $c \geq 10000$ messages (and consequently a much larger number of SCI packets) were transported over the network. Measurements involved a variety of topologies with both ringlets and switches. Operation was perfectly reliable between the AlphaServer 4000 5/300, AlphaStation 500/400, and EB66, but less so when RIOs 8061 were involved.

- The results indicate that one-copy implementations can improve asymptotic bandwidths by $\lesssim 15 - 35\%$ over what “thin” message passing libraries of the conventional kind (two-copy) can achieve. (Evidence for the relative cost of “thick” message passing libraries, such as MPI, is presented in section 3.3.3.

- Whether or not optional write gathering is turned on has a very strong effect on communication performance for the current generation of Dolphin SCI cards, even if messages remain small (see Figures 3.37 to 3.40).

### 3.4 Lessons learned from benchmarking

In hindsight, practical experience from taking measurements on a range of new hardware technologies [11] educated us about the following secondary effects, that emerged from our deliberate design choices:

- The message passing programming model puts shared memory architectures at a disadvantage, by requiring two-copy implementations; instead of one-copy or zero-copy implementations. In order to do distributed shared memory architectures like MEMORY CHANNEL and SCI better justice, the benchmarks have to be rewritten against other interfaces (refer to section 3.2.2), or extra effort has to be invested for arriving at quantitative estimates as for the size of this bias (refer to section 3.3.4).
• Our adaptation of message passing in general and a subset of MPI in particular induced the requirement for a reliable communication mechanism. The portable part of the relevant application-level code does not include any provisions for timeout handling. This requirement is for instance not met when the AAL5 (datagram-like) service is used for accessing ATM networks in Unspecified Bit Rate (UBR) mode. Alternative use of ATM's Constant Bit Rate (CBR) is able to settle the issue for this particular technology. However, the precision of a series of results that were obtained from ATM in UBR mode [11] was severely restricted by low repetition counts, as necessitated by frequent packet losses.

• By virtue of the same choice for message passing conformity, receivers pull for completion status, rather than awaiting calls to interrupt handlers. Thus some parameters are not directly translatable into the (paper) model of the ATLAS LVL2 trigger, which (somewhat arbitrarily) relies on an estimate of context switching times as one of its input parameters.

• The resolution of portable per-process timers (such as clock(3) in UNIX) is often poor in the order of tens of microseconds. We follow common practice and overcome problems arising from limited clock resolutions by aggregating large statistics from c rapidly repeated tries and by reporting averages. This helps to indirectly reveal times that are otherwise too small to observe. (It also helps to minimize the influence of the timing function's own calling overhead).

The method is imprecise insofar as some studied algorithms, together with their working sets, are sufficiently small to fit into the (first level) caches of today’s microprocessors. Repeated execution therefore tends to indicate lower execution times than would be observed on single runs, since starting from second cycles all instructions can be fetched from the cache. Practical implications for communicating algorithms are only small, however, because they include interference between communication and computing; i.e., in them progress is also bounded by the state of remote processes, and thus by (relatively slow) communication.

The resolution of real-time clocks is often superior, where available. Their use nevertheless entails other peculiar disadvantages. They report wall-clock time; therefore other processes’ execution times show up in the measurements, if taken on a time-shared environment, as well. The method of choice for reducing the amount of influence from myriad background activities is to repeat identical measurements at different times, when the time-shared system is obviously vacant, and then report the minimum reading as most meaningful result.
Chapter 4

Modeling as a tool for optimization

And now for something completely different.
Monty Python’s Flying Circus

4.1 Scope of modeling and simulation

In many situations generally, and as part of the planning process of complex systems in particular, it is interesting to know about the behavior of systems under the influence of various internal and environmental conditions and parameters [24]. This chapter investigates the use of mathematical and computer models as related tools for aiding such investigations.

The generic term system here applies to both physical systems (for instance digital circuits, production plants, ecological systems) and to logical systems (for instance air traffic regulations, interactions between members of a group, resource consumption in telecommunication networks).

The 2 rows in Fig. 4.1 on the next page illustrate the relationship between a dynamic system and its model. The drawing’s horizontal decomposition suggests a distinction between static and dynamic features, which are indicated as rectangles and arrows respectively. This distinction is similar in kind to those that are applied between computer hardware and software and between morphology and function.

The relationship between a system and its model must be such that both exhibit the same observable reactions when presented with corresponding sets of inputs. Unlike a black box, the model places requirement restrictions not only on observable reactions, but it is also concerned with internal representation (static aspects) and transformational rules (dynamic aspects) of the system at hand.

The internal representation requires the identification of a sufficient set of orthogonal state variables. Vectors of state variables describe all possible system-
Figure 4.1: relationship between a real system and its model

level states. Dynamic aspects are characterized by a set of time-dependent transformational rules, which act on state vectors and change some of their components to affect state transitions.

The most obvious and straightforward approach for obtaining information about system behavior is to bring about the desired internal and environmental conditions and then observe the behavior of the very system by itself. This is what happens in physics experiments, for example. The approach is nevertheless often infeasible because of potentially high cost in money and manpower, long time scales, mainly committed resources, and dangerous implications in case of failure, etc.

As an alternative, one can construct models, i.e. small-scale logical or physical replacements of systems. The models aim primarily at capturing the systems' function-critical essences and are free to neglect any peripheral details.

Determining what is essential is of course a non-trivial task, as has been pointed out in more general contexts by ancient philosophers already. Models that use computer software for representing state descriptions are advantageous, because they allow for a bootstrapping process by which initially trivial models can pragmatically evolve towards their systems' behavior through rapid cycles of re-adaptation, until the desired trade-offs between accuracy and complexity are achieved.

Experiments can then be conducted on a model rather than on the system itself. Because the model only approaches the "real thing", we call these token experiments for clear distinction. The approach may benefit from less stringent constraints, because token experiments can now be carried out in a well controlled virtual environment. In particular, models provide the opportunity to fail under controlled conditions.

There are more benefits to be had from the model's potential representation as software: a computer program can exhibit reproducible behavior with little effort (hitting the return key on a computer keyboard might do) and under exactly matching circumstances. It is also well observable and to some extent self-documenting.

The ability to use the mind for building scenarios and exploring ideas before choosing among competing options has been described as an important aspect
of human intelligence. It offers a position of great advantage as opposed to those creatures whose actions follow more directly from genetic or environmental causes, and for whom trial and error does not work primarily for the individual, but only on an evolutionary time scale. Speaking in very general terms, the motivation behind modeling in general, and the type of computer modeling that we are concerned with in particular, can be envisaged as a modest quest for tools to support the human brain in analyzing and planning activities and to perhaps extend its scope in some particular domains.

The same urge has contributed to the development of mathematics in the past. Its apparatus is adequate for compressing the complexity of problem domains into formulas that make up mathematical models, which can then be subjected to well-defined manipulations. Questions about a system can be answered, and aspects of its behavior can be predicted by “solving” the corresponding mathematical model.

For the sake of comparison with simulation, one can point out that mathematics allows users to cover a lot of ground, provided that they can handle the mathematical formalism well. But both the lack of intellectual capacity and of adequate mathematical tools often place limits on this approach, if systems of non-trivial complexity are concerned. In such cases we must be content with acceptable levels of confidence regarding the model’s correctness, and thus try to benefit from other, perhaps less formal, methods for establishing and confirming subtle cause-effect chains. The advent of high performance computers allows a complementary approach, by bringing considerable computing power to bear on a problem statement that is expressed in a slightly different form.

A distinction between mathematical and computer models cannot be drawn on the grounds of computer utilization alone, since computers are by now indispensable tools in many disciplines and support many kinds of techniques. What is characteristic for mathematical models is that they require transformational rules that are stated using only the laws of formal mathematics, so that the solution is given in terms of closed formulas. Analytic solutions can often be given only in principle, because the formal operations necessary to obtain them are too difficult or cumbersome to perform. In these situations numeric solution techniques for mathematical models or computer simulation are appropriate alternative tools. The principal advantage of the latter is its extreme flexibility (although this may lead to obscurity).

Both rely on incremental and iterative problem solving approaches. They repetitively manipulate the set of state variables via the formally stated transformational rules and thus “explore” the state space, rather than calculate the parameters of the “best track” that leads to the right solution all at once.

Thus computer models feature an additional degree of freedom in the description of state transitions, because they allow the inclusion of transformational rules in the algorithmic notation of programming languages. They also allow to perform checks on non-functional requirements, such as usability, performance,
cost, survivability, and reliability. Computer hardware and software tools are employed to build an interpreter, which captures the dynamics of the original system by imitation. By executing only those state transitions that are allowed by transformational rules, as stated by software, it restricts the system to states that would be enforced by what constitutes “nature” in the case of the real system.

The notion of virtual reality is by now well established and familiar to almost everyone primarily from the hype created around Hollywood gadgets and video games that take players into virtual worlds. It has also been argued that animal experiments will become unnecessary in the near future, because computer models of physiological processes may suffice instead. The recent Accelerated Strategic Computing Initiative (ASIC), whose aim is to deliver computers performing in the TeraFLOPS range in the U.S. within the next decade, has partly been described to the public as a secure means for obsoleting nuclear testing through computer simulation, or what is specifically called virtual weapons testing. Clearly those issues carry important non-technical connotations; however, they can readily serve to show that a link between computers and modeling is broadly recognized by now.

We are interested next in characterizing computer modeling in different potential roles, depending on whether it works as part of a methodology that operates top-down (deductive) or bottom-up (inductive). We associate the first primarily with applications in the natural sciences and observe that here the emphasis is on verifying assumptions about the inner working of systems, as revealed by model parameters. We associate the second mainly with applications in engineering, where the emphasis is mainly on inference, for learning about the behavior of large-scale systems and for carrying out design optimizations at reduced cost.

4.1.1 Role in top-down analysis

Contemporary leading-edge research in natural sciences is mainly driven by a reductionist agenda (thus the term top-down) that tries to come up with simple explanations for observable phenomena, which are in line with the few basic constituents and principal laws that we regard as (in principle) governing all of nature. According to the rules of the scientific process, observations and experiments either lend support to existing theories or lead to their rejection along with the formulation of new hypotheses. As we’ve noted before, modeling can come in at this point as a convenient tool for carrying out token experiments, which add to the number of scientific questions whose answers and consequences can be investigated.

Token experiments performed on models are similar to real experiments, but work on a logical rather than material basis. The corresponding computer program (in the case of computer modeling) forms a representation of a part of the physical world according to a particular theory; subjecting the theory to ex-
Experimental proof amounts to running the program with a certain set of input parameters. Observed differences between direct experimental evidence and the model's behavior in matching situations assist in its calibration via the fine-tuning of internal parameters.

Practical experience suggests enough obvious reasons for not trusting results from computer modeling as full replacements for real experimental evidence alone. A model is a system’s functional substitute, which has been crafted from the original in a process that is potentially error-prone and requires debugging. The complexity of the software process also adds new sources of error: things can go wrong, which would not have happened in real life. Perhaps because of the discrete nature of the quantities involved in digital computer programs, even small defects can have unproportionally large and potentially negative overall outcomes.

There are nevertheless important application areas where only this next-best approximation of empirical results is accessible. Theories of cosmology or biological evolution for instance involve time scales that are clearly prohibitive against the notion of controlled real-world experiments; yet successful — or at least generally accepted — computer models exist in these domains.

Pointing out limitations of computer modeling therefore does not discredit the method as a whole, but rather encourages a pragmatic view, where it is seen as one tool out of several from a set of related techniques, each of them implying different pros and cons.

Consider modern meteorology as a discipline that relies to a large extent on techniques for the numerical solution of mathematical models as abstract representations of atmospheric processes. Although people would usually agree that the weather forecast provides a useful set of services, the uncertainties of its predictions are known to everybody from trivial experience, and people know how to account for this in their daily lives.

### 4.1.2 Role in bottom-up designs

In many engineering disciplines modeling has gained broad acceptance as a constituent part of bottom-up design processes of complex systems, where it complements other design stages, such as analysis, specification, prototyping, and testing.

For example, architects usually provide small-scale physical models of future buildings, which help customers in interpreting design proposals and comparing them to their own requirements. Models are designed to have better appeal to faculties of human imagination than other abstract planning tools. They provide representations that approach the external qualities of final systems in their typical surroundings, yet at reduced scales and complexities.

Modeling can help to reveal subtle design errors, which may otherwise be hard to spot in “two-dimensional” plans. Its exercise allows elimination of dark
corners of systems’ desired behaviors and facilitates trade-off decisions in the face of conflicting requirements, because diverging options can be played out.

While the indicated architectural models are completely static, car manufacturing and aeronautic industry, for example, harbor applications for dynamic models as well. These study functional properties of prototypes that exemplify various candidate designs, in scenarios such as wind channels of provoked car crashes.

Both static and dynamic modeling have recently benefited from the utilization of modern computer technology. For instance, architects now commonly include virtual walk-throughs in their presentations, i.e., computer generated movies that show the shapes of buildings on computer monitors in realistic perspectives and color shadings from arbitrary views, very much as they would be observed on real visiting tours. (Similar graphical resources are available for exploring the proposed underground infrastructure for the ATLAS experiment.) If sufficiently powerful computers can be put to work, image generation may proceed in real time. Visitors then get full control over their movements via special sensory equipment, which empowers their normal human senses in pre-fabricated virtual environments.

Modeling fits in halfway between an abstract specification of the planning phase and the concrete implementation of the construction phase. As such, it can attempt to combine advantages from both ends.

An abstract specification is flexible and potentially cheap in the sense that only a limited amount of resources needs to be committed to its production. It provides a description of the problem at hand in a form that fits some established theory, to which a well known set of rules can be applied, and which can potentially be subjected to formal proof (thus checking proper mechanical layout, for instance).

The description is typically stored in a form (a formula on a blackboard, a blueprint of an architect’s plan, a file on hard disk), which makes it easy to copy and distribute its most up-to-date versions among participating parties. It facilitates cooperation, work sharing, and early error detection and helps to amortize design costs as familiar patterns of work reappear.

Modeling allows the exploration of different scenarios and comparisons between competing options, including differently parameterized versions of the same system. A designer of a system in which various components interact may for instance want to know its behavior if certain components fail to function. If this reveals specific weak points in design, he can either re-parameterize (e.g., choose “larger” parameter values, and thus provide extra fault tolerance through redundancy) and then re-evaluate the model, or acknowledge the presence of stronger requirements and reiterate to earlier planning stages, thus taking the system effectively back to the drawing board.

1http://sgvenus.cern.ch/VENUS/atlaspage.html
A well-structured engineering design process is a sequence of planned activities, which leads from initial requirements to a matching implementation. Although the order of steps needs not be entirely linear, but may include feedback loops, where findings at later stages (such as modeling) lead to backtracking and repetition of work after modifications at earlier stages (such as specification), these deviations are costly and should be kept local between adjacent stages, or at best entirely avoided.

The abstract nature of models and the availability of computers for repeating mundane tasks significantly reduces the cost of recreating models in some cases: a movie sequence for a virtual walk-through can be regenerated in a fully automatic process, if the building’s floor plans have been prepared in a sufficiently abstract form, via Computer Aided Design (CAD). But some costs (such as those for computing time) still remain, and the argument also generally fails for instance in cases where computer models have to be created through laborous manual intervention (such as code generation).

The very reason for identifying discrete stages in the design process is to reduce the amount of redundant work and to perform correct steps at the right times. We’d like to stress this point by suggesting that if the methodology is bottom-up rather than top-down, the model is a temporary tool rather than a subject of investigation. As far as efficient work is concerned, its self-study should strictly be confined to the necessary amount of debugging alone. This observation leads to different consequences, depending on whether one regards models that qualify for typical use in natural sciences (top-down) or engineering (bottom-up).

While in the former case one wishes to learn about a system’s parameters, constituents, and interactions, and dissecting a successful model might be a useful way to partly reveal them, in engineering one works from a different set of rules. If the system’s specification is not sufficiently detailed or stable for its abstract representation to be achieved in a format that fits modeling, the modeling exercise must be delayed.

System designers and modelers might constitute different groups of persons. Since an important aspect of modeling is to protect against subtle weak points in designs, the first might not always be willing to let modelers pry over their shoulders, or make detailed documentation available at early stages. Such communication is especially vital with regard to those semi-random design decisions that are always involved in planning to some extent. Engineers may oblige to comply when forced by hard external milestones, such as delivery dates, but less so when faced with internal requirements that originate from an organizationally separate modeling exercise, because this leaves more head room for ad-hoc design decisions at conveniently late times.

Modeling can tolerate some unknown features by introducing extra free parameters. Other open choices result in more substantial efforts that have to be repeated whenever alternatives change. If modelers cannot rely on the same knowledge that was accumulated during the basic design of the very system,
then their model’s behavior must obviously differ from what the system shows, and its verification must fail. Thus if “blind” guesses prevail during a model’s creation, then the modeling exercise serves no other than educational purposes; in particular, the model cannot serve as the system’s logical substitute in token experiments.

In some application domains (such as chip design) a better solution can be approached by making use of specialized ready-to-use simulation programs. They are positioned to appeal to design engineers by being tailored to narrow tasks and by featuring intuitive controls, for instance. Thus they reduce the amount of tool-specific knowledge that needs mastering, and allow experts in the application domain to carry out simulations without having to rely on software engineers.

Non-custom systems, such as those often found in research environments, on the other hand do not allow for such comfort, but require actual programming. Programming languages for simulation, such as MODSIM-II [16] (refer to section 4.3.1), somewhat reduce the discrepancy between required application-specific and tool-specific knowledge (and the required effort), by covering low-level considerations that relate to the organization of the modeling task. But they still leave a large emphasis on tasks that may be foreign to the application at hand.

### 4.2 Discrete event simulation

The evaluation of computer models involves a progression among system states via the manipulation of state variables. This is effected by the time-dependent execution of transformational rules from models’ descriptions. In order to guarantee the right sequence of iterative steps, a central control algorithm is required, which guards the flow of simulated time (also known as simulation time), such that the right rules are triggered at appropriate times. In the following we are concerned with possible ways of implementing simulators with proper control mechanisms.

Depending on whether the transformational rules of the model bring about small changes \( dx \) to state variables during infinitesimal times \( dt \) (as a function of \( t \)) or whether these changes are by arbitrary large amounts \( \Delta x \) that happen only at precise times \( t \), a distinction between continuous and discrete models can be drawn. Digital computers are appropriate tools for developing and evaluating the latter. The present discussion therefore focuses on discrete models.

Some extra pieces of terminology are commonly applied; they are also useful for arriving at a full definition of all terms that contribute to the clause discrete event simulation.

**Epochs** \( t_i \) are those points in simulated time when state transitions occur, they are measured in base units of simulated time \( t \). The theoretical importance of real time, as opposed to simulated time, in a dynamic computer model is only secondary and restricted to the (nevertheless practically important) concern
that the model’s evaluation must obey reasonable limits of resource consumption, including that of CPU time. One can imagine simulated time as having an entirely different dimension from real time, as Fig. 4.2 effectively suggests by plotting both along 2 different orthogonal axes. This already indicates that comparisons between epochs and values of real time are usually inappropriate. As an exception from this rule, the ratio between progression of simulated time versus real time can serve as a crude metric for how efficiently a particular computer serves in the evaluation of a given model.

*Events* \(e_i\) mark instantaneous changes to values of 1 or more state variables at epochs \(t_i\). They are specified as fragments of procedural code in some programming language. The distinction between process-oriented and event-oriented discrete event simulation (see below) mainly deals with how these fragments can be arranged into large, yet suitably maintainable, programs. The notion of events is also useful for highlighting a difference between real time and simulated time: while the computer executes the instructions forming one event, real time progresses, but simulated time stands still.

The model’s dynamic behavior is completely determined by a list of pairs \((t_0, e_0) \ (t_1, e_1) \ldots \ (t_n, e_n)\). Only few elements of the list are known when the model’s execution commences, for events \(e_i\) are free to delete or generate new pairs \((t_j, e_j)\), provided that \(t_j \geq t_i\). (This rule expresses that we don’t want “paradox” events with effects on their own pasts.) The control mechanism must bring about the state transitions caused by events \(e_i\) in the correct order, for tracking the model’s evolution over time; i.e., \(t_0 \leq t_1 \leq \ldots \leq t_n\) must hold, if the control algorithm picks events from the list in left-to-right order. Two approaches for achieving this are common.
4.2.1 Clock-driven approach

In a clock-driven approach the control algorithm increments the simulation-time counter repetitively by a constant interval unit $\Delta t$. All events $e_i$ that are eligible to occur during the intervening time interval, due to $t_i \leq t < t + \Delta t$, are picked and the transformations required by $e_i$ are performed. The advantage of this scheme is its inherent simplicity, which maps well onto straightforward implementations in a wide range of programming languages. Its disadvantage is that in order to execute events in the correct sequence, one has to guarantee that time values are discernible, and thus choose a sufficiently small interval unit $\Delta t$, such that $t_i, t_j \in [t, t + \Delta t]$ implies $t_i = t_j$, for every $t$. Ticks have to be short in comparison to response times inside the system.

Normally this places a severe performance penalty onto simulations, because simulated time progresses only slowly. If events are unevenly distributed in simulated time, its granularity is too fine during periods where events occur only sparsely, and the control algorithm thus spends a lot of time in scanning “empty” time intervals, where no state transitions occur at all. Scanning is a process of the order $O(n)$, where $n$ is the number of events in the model. Apart from trivially sized applications, the scheme thus only provides net advantage, if all epochs are whole multiples of a given clock cycle $\Delta t$, such that $t_i = i\Delta t$, as it can be expected in the case of models of clock-synchronous digital devices, for instance. Combining a model with that of another device that runs at a different clock rate would require the choice of the greatest common denominator as new $\Delta t$. If the value is much smaller than any of the contributors’ (consider cases where prime numbers are involved, for example), then the combined model will perform with unreasonably worsened efficiency.

Within the scope of preparational work for the ATLAS experiment, a clock-driven approach to simulation was chosen for a model of an ATM switching network (refer to section 3.3.2). Because both this model and a version of the SIMDAQ framework (refer to section 4.3) used C++ as implementation language, a proposal was made at some time as to merge these 2 models into one. Although the catchword C++ promised automatic interoperability to many, the attempt proved impractically hard, because the framework used an event-driven approach instead (refer to section 4.2.2). The point we want to make here is that it is worthwhile to understand the different implementation choices, because these have an important discriminating effect, which may dominate in relative importance over what is caused by using different programming languages, for instance.

4.2.2 Event-driven approach

The event-driven approach features a mechanism that is more mature in comparison to its clock-driven counterpart, because it determines the best particular unit interval at each new epoch value $t_i$. To achieve this, the records are kept
in a sorted list, termed *event queue*. The control algorithm repetitively picks an element \((t_i, e_i)\) from the head of the queue, sets the simulated time to \(t_i\), and executes the procedural code that is associated with the event \(e_i\).

This arrangement makes steady efficient execution of the model less vulnerable against periods of simulated time where events are sparse, but has consequences on the possible styles of implementation in general-purpose programming languages. The event queue is essentially a data structure that contains tagged time stamps \(t_i\) (represented as values from some simple numeric type) along with identifications of code fragments that implement \(e_i\) (second order types). This requires the concept of pointers to code, for otherwise implementations involving long if-cascades have to be tolerated (state machines).

In programming languages that are not object-oriented (e.g. C or Fortran) these identifiers can be represented as pointers into procedures (where available). These mediate efficient and sufficiently flexible implementations of event queues, but also cause important disadvantages, since they represent only starting points for execution (possibly along with a set of actual parameters). Unlike closures (for instance in Common Lisp [70]), they do not convey a more complete description of inner state. Therefore every state variable, whose value is requested or updated by statements in \(e_i\), either has to be copied as an actual parameter in the procedure call or else accessed as a global variable.

They are furthermore only usable under severe restrictions, because the necessity to store several events in a common event queue calls for a mandatory common procedural interface for all events. Since, in particular, the same number and types of formal parameters are required, this often leads to solutions where only a single composite parameter is chosen, which at different invocation times points to structures holding actual parameters of different types. This infringement of strong typing is undesirable from a software engineering point of view. The second option is even more undesirable, because it strongly violates the important principles of encapsulation, modularization, and separation of concerns.

Object-oriented programming languages (such as Ada 95, C++, Eiffel, Java, and MODSIM-II) offer better solutions in this regard, because they allow a compartmentalization of all state variables into distinct sub-sets, with well-defined interfaces between otherwise loosely coupled modules. State variables from a given sub-set can be suitably represented as attributes of a specific class. If pointers into procedures reference (static) member functions of classes (rather than global functions), these pieces of code may still access state variables as if these were global, yet from an outside view their visibility is firmly limited to the scope of class declarations; the globalness of state variables is thus suitably restricted to a degree by which events can manipulate only relevant sub-dimensions of state space.

Thus the object-oriented paradigm allows to express fundamental requirements for modeling in a natural way. This observation is consistent with the fact
that Simula 67, an early programming language for discrete event simulation, was the first language which included facilities for describing objects in its type system.

The paradigm suggests that real-world entities are mapped onto objects and defines the concept of classes for accounting for the fact that entities usually fall into discernible categories, whose overall number is small, and which form hierarchic relationships with each other. Unlike the preceding paradigm, which employed algorithmic rather than object-oriented decomposition, this view emphasizes both static (hierarchic dependencies and inheritance) and dynamic (controlled interactions via exchanges of messages) relationships between objects, rather than sequential relationships between algorithm steps alone.

4.2.3 Process-oriented discrete event simulation

The object-oriented paradigm allows direct mappings between structural relationships in the real world and the topology of syntactic constructs, as provided by programming languages. For modeling purposes a similar correspondence is also desirable with regard to dynamic aspects.

This can be achieved by a form of notation that groups causally linked events, which also relate to close points in simulated time, such that they appear next to each other in the program's source text. The resulting approach is called process-oriented discrete event simulation, with processes rather than events now constituting the main conceptual building blocks.

It can be argued that, while object-orientedness adds locality of state (the vertical axis in Fig. 4.2 on page 152), process-orientedness provides locality of simulated time flow (the front horizontal axis in Fig. 4.2).

A process constitutes a newly invented kind of procedure, which can characteristically consume simulated time. It consists of a number of events, which update state variables, plus a number of prescriptions for the advancement of simulated time between these events. Inside events it uses only conventional programming language statements, whereas in the intermediate sections it additionally relies on new syntactic constructs, such as \textit{wait}. In addition to waiting for a fixed amount of simulated time to elapse, suitably adapted programming languages also offer versions of \textit{wait} for resuming process execution on signaling of custom events.

\textit{wait} statements allow a form of notation where processes can be written as linear code fragments, such as \{e_1; \textit{wait} (t_4 - t_1); e_4\} and \{e_2; \textit{wait} (t_3 - t_2); e_3\}. This is preferable over an event-oriented notation, which relies on more numerous and scattered pieces of tagged code, such as \{t_1 \rightarrow e_1\}, \{t_2 \rightarrow e_2\}, \{t_3 \rightarrow e_3\}, and \{t_3 \rightarrow e_4\}.

The first form conveys information about causal relationship directly from the local source code's composition. In the latter case fragments can be arbitrarily
distributed (events can be syntactically mixed in any order): relationships are only established at run-time, when tags yield similar values.

Process-oriented discrete event simulation is thus clearly desirable from the human reader’s point of view. It requires a programming language (or run-time environment) that includes the concept of multiple parallel threads of control, either in the form of coroutines or tasks (threads).

Assuming that a linear sequence of events is implemented as a coroutine, the rules for evaluating \( e_1; \text{wait} (t_4 - t_1); e_4 \) can be laid out as follows: execute the procedural code for \( e_1 \), then suspend this process’ execution until simulated time reaches \( t_4 \), and give other processes the chance to catch up in the meantime, finally execute the procedural code for \( e_4 \).

Coroutines have weaker expressive power than tasks. They provide only support for non-preemptive parallel programming (context switches between tasks are restricted to explicitly prepared situations). Unlike with tasks, where the time flow of suspend and resume operations is beyond the control and responsibility of applications (preemptive multitasking: context switching between processes can be enforced by the operating system at any time), coroutines require provisions for explicit passing of control in the form of reserved program statements.

Coroutines form natural correspondences with simulation processes, because all points where control may potentially pass in the latter can easily be identified by enumerating \textit{wait} statements. An event \( e_i \) is ready to resume control on behalf of its process when it becomes due in simulated time, that is \( t_i \geq t \). A global scheduling mechanism can rely on re-evaluating these expressions whenever \textit{wait} statements are reached, because simulated time \( t \) remains constant at \( t_i \) throughout the evaluation of single events \( e_i \).

Committing to a general-purpose programming language (such as C, C++, or Fortran) alone does not yet settle any issues relating to the generic modeling exercise, because their raw semantics are not rich enough (by the very definition of general-purpose programming languages) to provide abstractions for enabling process-oriented discrete event simulation. Their use for this specific purpose requires an educated commitment to an alternative of syntactic form. Thread libraries (e.g. POSIX threads library [51]) may be used to bridge the semantic gap and add event-level parallelism and suspend/resume semantics to a range of host languages.

### 4.3 SIMDAQ — A generic model of the ATLAS LVL2 trigger

SIMDAQ [44, 45] provides a generic framework for embedding models of different DAQ components and has been used as tool for modeling asynchronous versions of the ATLAS LVL2 trigger. The characteristics of synchronous systems do not
lend themselves well to discrete event simulation; it is more straightforward to study them with other (mathematical) tools.

SIMDAQ’s original implementation was built on top of the programming language MODSIM-II [16] (refer to section 4.3.1). The motivation for providing a tool, in addition to what is given by a programming language for discrete event simulation alone, was mainly based on the following thoughts.

- By establishing a set of rules, which are in line with assumptions and terminology agreed on in the ATLAS community, we hoped to arrive at a situation where different models yield comparable results.

- The provision of reusable code libraries reduces the overall effort that has to be met when producing new models relating to several technological choices.

SIMDAQ adds to the lower MODSIM-II layer’s functionality in the areas of configurability, support for generic objects for describing the building blocks of processor farms and switches, input of simulated physics events, and collection and presentation of results.

It treats processors as very abstract entities: only 2 states matter, namely calculation (when they do not influence other entities) and communication (when they do). The transition between these states is mainly governed by stochastic data on traffic patterns and sampled execution times, as available from physics simulations and algorithm benchmarking.

Simulation runs for the ATLAS Technical Proposal [5] were based on sets of simulated physics events; these events were available from GEANT code. After feeding them through a simulated LVL1 trigger, they were stored in a portable text format. This type of event generation has the important advantage of yielding realistic correlations between event fragments, as opposed to what a randomizing event generator can achieve. Data sets were in the order of 10000 physics events (with multiple random repetitions); this corresponded to data arriving from LHC during 10 ms in the life of the real ATLAS detector.

The syntactic description of the portable text format has been made available in a formal specification. Starting from this, code for extracting event information from portable text files was generated automatically for several programming languages. This proved valuable, because it allowed several groups to access shared files. The groups used different software tools and worked on algorithm studies and architectural modeling.

Descriptions of the hadronic and electromagnetic calorimeters as well as of the Transition Radiation Tracker (TRT) were contained in the model that produced the quoted set of results. The segmentation of sub-detectors was chosen in agreement with the number of readout crates, as known at the time. This gave 8 plus 32 LVL2 readout buffers for the calorimeters and 32 LVL2 buffers for
the TRT. Provisional assumptions had to be made on the grouping of fibers into

The model produced graphs showing distributions for occupancies of LVL2
buffers and processors, as well as latencies that accumulated while LVL2 de-
cisions were pending. Calculations were performed for both setups with ATM and
SCI switches, and results were reported in [5]. A detailed description of a re-
presentation of the C104 Asynchronous Packet Switch that fits into SIMDAQ’s
framework is given in section 4.3.2.

Although the achieved results were not yet considered as indications for actual
design choices (the model was still too “generic” in many areas), they helped to
demonstrate the suitability of the used approach in general.

In addition to discrete event simulation, “paper models” were established
as well. These are the next step up from back-of-the-envelope calculations, by
virtue of using electronic spreadsheets as their substrate. They yield first-order
estimates, by combining average values for parameters such as expected multi-
plicities of trigger menu items and data volumes per sub-detector. In contrast,
discrete event simulation works on an event-by-event basis, considers stochastic
distributions of parameters rather than averages, and is able to take network
contention and queueing into account.

4.3.1 MODSIM-II

MODSIM-II [16] is a proprietary, block-structured, and object-oriented pro-
gramming language for process-oriented discrete event simulation, which borrows most
of its basic syntactic structures from MODULA-2 [79]. It adds support for object-
oriented programming and process-oriented discrete event simulation.

For the second purpose the language defines special kinds of procedures, called
WAITFOR and TELL methods (synchronous and asynchronous processes respec-
tively), which can consume amounts of simulated time during their invocations.
Ordinary procedures — ASK methods in the language’s terminology — consume
only real time. An inherent scheduler allows for any number of processes to
execute concurrently relative to simulated time.

Control potentially passes between methods whenever WAIT statements are
encountered. These can await the following changes: passing of certain amounts
of simulated time \( \Delta t \), completion of synchronous calls to time-consuming meth-
ods, and (re-)availability of resources from shared pools. MODSIM-II programs
must explicitly collaborate with the coroutine scheduler. This support of paral-
lelism is again similar to what is found in MODULA-2. TELL statements lead to
asynchronous method calls and may add to the overall number of quasi-parallel
processes.

MODSIM-II was used as implementation language for the original version of
SIMDAQ, which produced results for the ATLAS Technical Proposal [5]. Its role
was challenged in a later discussion that focused on alternative language choices
[43], but entirely excluded higher-level approaches based on existing libraries and tools (many of them available in the public domain).

The 2 competing (raw) programming languages can be characterized as follows, relative to the application at hand.

- MODSIM-II provides immediate support for discrete event simulation and convenient run-time error checking. Much existing work was done based on its use.

- Its lack of debugging support was felt badly during development. Programs were prone to slow execution and also suffered from more fundamental flaws in the language’s design, such as those relating to its rather unsatisfactory implementations of multiple inheritance and generic classes, for example. Since its compiler is available only from a single vendor, the language clearly cannot benefit from the rapid dynamics that help to shape and improve industry standards.

- C++ has obvious advantages in the area of execution speed, tool support (as well as tools’ prices), and availability of trained manpower; these benefits are well understood throughout industry.

- The add-on-library for modeling in C++, which was developed from scratch, was not faithful to the process-oriented simulation paradigm, by failing to incorporate mechanisms for event-level parallelism (refer to section 4.2.3). It invited proliferations from standard use, because it had to rely on voluntary adherence to semantic conventions alone.

### 4.3.2 A model of the C104 Asynchronous Packet Switch

This section presents a description of a particular networking technology that was incorporated into the SIMD-AQ framework. Its selection for highlighting is motivated by the following features.

- It produced results that were in good agreement with measurements that were taken on physical devices; thus its empirical validation succeeded.

- It was partially implemented in 2 version that differed in their amounts of applied simplifications. This redundancy provided some evidence as to the proper level of abstraction, in the case of a particular modeling exercise.

The C104 Asynchronous Packet Switch [68] from inmos SGS-Thomson is a full $32 \times 32$ crossbar for bidirectional routed communication. Communication links run the serial DS-link protocol that was originally introduced with T9000 transputers. It offers reliable transmission, transparent flow control, and support for arbitrarily large packet sizes.
A C101 Parallel DS-link Adapter is available for bridging into networks that run the DS-link protocol suite. Although T9000 transputers split all packets into 32-byte fragments, for enabling numerous multiplexed virtual channels over fewer physical channels, the C101 chip does not enforce this convention. (Each T9000 transputer has built-in hardware support for multitasking. This offers similar enhancements in the computation domain as virtual channels offer for communication.) The C101 serves as a protocol converter between a parallel interface with simple handshake and the serial DS-link interface; it allows devices other than transputers to communicate via a network of C104 switches.

The C104 has the following routing characteristics (numbers are quoted from the manufacturer’s data sheet [68]):

**Wormhole routing** allows the chip to send a packet’s head on one link while it is still receiving its tail on another. The start of activities on both links is delayed by only a brief switching latency of \( \sim 1 \mu s \).

**Interval labeling** allows a simple scheme for arriving at routing decisions while still granting enough flexibility for describing a variety of network topologies. Each of the 32 outgoing links has a table of up to 36 non-overlapping address intervals associated with it. Sets of intervals for describing grids, cubes, and other topologies can be constructed by known algorithms. Addresses are specified as 1- or 2-byte integer values. The fact that tables are used on a per-link basis allows for a segmentation of a given network into multiple (non-overlapping) logical units.

**Group adaptive routing** is a mechanism for increasing communication bandwidth beyond the 100 MBit/s that are normally available on each link (\( \sim 10 \) MByte/s after the DS-link protocol’s overhead has been taken into account). It causes the C104 to utilize another member from a group of consecutive out-bound links, whenever a requested link is busy.

**Universal routing** instructs the chip to route packets to random intermediary destinations, where **header deletion** then restores their original addresses. This two-stage routing scheme has been shown to increase overall network throughput under conditions of heavy load. It makes packets bound for the same destination use different paths, thus it effectively distributes peak loads over redundant network resources.

Modeling the C104 benefited from the fact that the activity depended to a lower degree on other information, in addition to what is provided in the manufacturer’s data books (refer to section 4.1.2). The behavior of programmable components (nodes) depends largely on their specific embedding and programming. Relevant design work (some of it outside CERN) is still going on, and not all parameters have been fixed (or documented) yet.
The present model’s qualitative level of detail is such that it faithfully reproduces the chip’s wormhole routing, interval labeling, and group adaptive routing characteristics. Universal routing, header deletion, passing of DS-link protocol tokens, and data buffers (≤ 100 bytes capacity per link) are not taken into account. This design was chosen with the aim in mind of keeping the model simple and efficient.

A more detailed model could additionally include a representation of a larger number of building blocks, such as it is indicated in Fig. 4.3 on the following page. Here each link of a C104 switch is viewed as a pipeline of 2 or 3 building blocks, depending on whether the link’s capacity for carrying incoming or outgoing traffic is concerned. Building blocks have the following characteristics in common. Each building block can accept input and emit output at varying speeds; it can also introduce delays (between receiving and forwarding tokens) and store arriving tokens in an internal buffer (while its input speed exceeds its output speed). Special building blocks act as protocol converters (gateways) and data sources or sinks.

According to this level of detail, the chip has a buffering capacity of tens of tokens (bytes) on each of its internal paths (input link to output link). Its core is clocked at 50 MHz, therefore the crossbar at its center runs at a higher speed than what is obtainable on any of its links.

MODSIM-II’s object-oriented features allow for a natural representation of building blocks and their varying details (amount of delay, available buffer storage space) in the consistent manner of related classes.

The labels that appear next to arrows in Fig. 4.3 form a flow control protocol for negotiating speeds of token transfers between individual building blocks. The protocol tokens allow senders to inform receivers about starting and finishing packet transmissions. It allows receivers to hold and release traffic from senders. This specific convention allows for an efficient simulation, because states need updating only when packet transmissions begin or end, or when collisions occur.

This desirable characteristic is lost if a representation of the C104’s own flow control protocol is introduced into the detailed model. According to its rules, senders must wait for the arrival of special flow control tokens (FCTs), before they may emit further tokens to receivers. The protocol would obviously slow the progress of the simulation, especially in situations where long packets prevail, because it requires state transitions after short and reoccurring intervals.

A prototype version of the detailed model has been implemented. However, in the following we refer to another version of the model that is comparatively simpler, and whose good results allowed us to positively account for the following aspects.

- The detailed model incorporated ambitious treatment of delays that were in the order of magnitude of tens of nanoseconds. Results from first measurements in the testlab showed that some other equipment that was used
Figure 4.3: schematics of a detailed C104 model

<table>
<thead>
<tr>
<th>topology</th>
<th>switches</th>
<th>connections</th>
<th>worst case</th>
<th>bisection</th>
</tr>
</thead>
<tbody>
<tr>
<td>ring</td>
<td>$N$</td>
<td>$N$</td>
<td>$N - 1$</td>
<td>$2$</td>
</tr>
<tr>
<td>$d$ dimensional grid</td>
<td>$N$</td>
<td>$dN$</td>
<td>$d\sqrt[4]{N}$</td>
<td>$\sqrt[4]{N}$</td>
</tr>
<tr>
<td>arity $d$ Omega network</td>
<td>$N \log_d N$</td>
<td>$(d \log_d N) / 2$</td>
<td>$\log_d N$</td>
<td>$N$</td>
</tr>
<tr>
<td>arity $d$ Benes network</td>
<td>$2N \log_d N$</td>
<td>$d \log_d N$</td>
<td>$2 \log_d N$</td>
<td>$N$</td>
</tr>
<tr>
<td>crosspoint</td>
<td>$N^2$</td>
<td>$N^2$</td>
<td>$1$</td>
<td>$N$</td>
</tr>
</tbody>
</table>

Figure 4.4: scaling properties of different network topologies
in conjunction with C104 switches (such as C104/C40 interfaces for example) introduced delays that were by many orders of magnitude larger in comparison. This amount was initially unexpected. Since the C104 model will finally serve as a module in a simulation of a computing system at large, a pragmatic choice was taken as to restrict its level of detail (and thus complexity) to characteristics that will have an observable effect in the final system.

- Apart from this specific concern, we'd like to draw attention to the fact that a variant of Amdahl's law governs these issues: a highly detailed sub-model only makes sense if it is met by a similar level of detail and accuracy in other parts of the system. While for some sub-models their level of detail can be raised almost arbitrarily, given one is willing to invest enough effort (typical example: well specified hardware device), this is not true for others (typical example: processor with multitasking operating system).

- We were restricted to a single informal source of information [75] that explained the decomposition of C104 asynchronous packet switches into building blocks. Building a large programming effort on relatively sparse information did not seem advisable.

- The aberration caused by the simplified model's ignorance of buffering clearly shrinks in importance as packet sizes increase. Packets traveling the local switch (ROI fragments) are typically in the order of magnitude of several kBytes, and thus large compared to the switches' internal buffering capacities. (Note however, that T9000s split all packets into chains of 32-byte segments.)

- An evaluation of results that were obtained with the simplified model indicated that the chosen level of detail (accuracy vs. required effort and resource demand) was appropriate for our purpose.

Apart from a representation of individual switches, the C104 model is also concerned with network topologies, i.e. the aggregation of switches into networks of different arrangements. Topologies offer a choice of trade-offs between required investments and path lengths through the network (as measured in number of hops).

Fig. 4.4 (reproduced from [58]) measures required technology-specific investments, in numbers of switches and their interconnections, as functions of network sizes. The free parameter $N$ describes the maximum number of sources and destinations that may be connected to the network, before it needs extension in size (additional ports).

Path lengths are indicated by upper limits on the number of switches (hops) that must be traversed between any combination of sources and destinations.
(For fat trees this worst case scenario e.g. implies that all messages must go
to the top of the tree, before they can be routed towards destination nodes
on their ways down.) Apart from cost issues, topologies also assign different
relative importances to concerns of electrical implementation, wiring efficiency,
mechanical manufacturing, and deadlock avoidance.

For instance, rings show linear scaling in both aspects that relate to cost and
latency respectively. Other topologies typically achieve reductions on the latter
by tolerating moderate increments on the first. Rings are inherently simple but
vulnerable against malfunctioning single links.

The simplified model enables 2 specific topologies: hypercubes have been
chosen because they are well covered in literature and are reasonably complex;
therefore they allowed us to test the model's handling of topologies in non-
trivial arrangements. The topology of GPMIMD (see Fig. 4.6 on page 166),
on the other hand, reflects the characteristics of an existing experimental multi-
processor, which uses both C104 switches and T9000 transputers. This provided
opportunities for comparing the model's behavior against what has been found in
laboratory measurements. We considered an enabling of this step as a necessary
contribution to debugging and calibration.

A more flexible scheme for enabling different topologies could alternatively
extract and reuse information from Network Description Language (NDL) files.
Transputers typically operate in run-time environments that require matching
up-to-date software descriptions of actual hardware setups in NDL format. This
flexible configuration mechanism is not enabled in the existing code for the follo-
wing reasons: NDL's Occam-like syntax [41] does not lend itself well to treatment
with tools for automatic parser generation. (This mostly relates to the language's
rather unorthodox use of whitespace for marking syntactic hierarchies.) In ad-
dition, commonly available tools such as lex(1) and yacc(1) are more readily
prepared for interfacing to programs that have been written in C, as what can
be achieved at best for MODSIM-II. Existing tools for processing NDL source
files produce their outputs only in undocumented (and thus unintelligible) binary
formats.

The C104 model includes code for constructing representations of fat hyper-
cube networks from the parameters \( N, l, \) and \( a \): a \( d \)-dimensional hypercube (see
Fig. 4.5 on page 166) consists of \( 2^d \) network nodes (C104 switches in this context),
where \( d = f(N, l, a) \). \( N \) is the number of outside nodes that require separate con-
nections to the network; \( l \) specifies the available bandwidth for network-internal
connections in number of group adapted physical links between every pair of
neighbors. We use the attribute fat to refer to hypercubes that use this C104-
specific feature as a means of enhancing throughput. \( a \) is a parameter that
helps to describe setups where electrical or mechanical considerations impose
constraints on the maximum number of outside nodes per C104 chip.

The code uses an iterative procedure in order to determine \( d \) from the given
parameters. It then records the required physical interconnections between those
C104 switches that are at hamming distances one apart, imposes an addressing scheme for outside nodes, and stores proper values into all address interval registers. Node addresses start from 0 and are consecutive; they are assigned in groups of $\min(32 - ld, a)$ addresses each, for nodes that connect to given C104 switches. Apart for outside connectivity to nodes, each C104 switch uses $ld$ out of its 32 links for establishing connections to neighboring switches. Some links may remain unused; address interval registers are set up such that these links cannot be used without provoking indications of error.

The model adheres to SIMDAQ’s conventions for dynamic relationships among objects and classes. In particular, a class `SimpleC104Switch` serves as an abstraction of a given network (the word switch is used here in an untypical way, for denoting networks rather than chip- or board-level entities), which hides its other constituents inside.

`SimpleC104Ports` provide a flexible and SIMDAQ-compatible interface between the network and different types of external nodes. By suggesting a fixed protocol between pairs of nodes, SIMDAQ can allow for varying types of nodes and switches to be combined in a transparent fashion. The mechanism requires matching pairs of ports for each connection between a network and terminal node. One peer is marked as owned by each participant by means of enumerating it in the owners `ChildQueue`. The binary relationship between ports and the directionality of the connection are conveyed by the settings of their fields `FarPort` and `IODirection` respectively.

The following standard protocol is used for sending packets over the network: the sending node accesses its port in `ChildQueue` and calls its `WAITFOR` method `SendMessage` with a parameter of type `Message`, which contains fields for identifying packet source, destination (both as `SimpleC104Ports`), and size. The method returns only after the packet has been sent off entirely and successfully (some simulated time passes in the meantime). When the packet’s transmission is complete, the simulation calls the receiver’s `TELL` method `MessageArrived` for conveying that it may immediately use the packet’s entire contents. Since the methods `SendMessage` and `MessageArrived` are defined in the abstract class `Node`, which resides close to the top of the static class hierarchy in SIMDAQ, objects from a wide range possess capabilities for acting as senders and receivers.

The simplified model tries to catch the behavior of the C104 chip by referencing only 2 parameters. The first describes the chip’s switching latency; the second describes its line speeds. The total time for sending a packet is calculated by adding the overall latency (accumulated over the entire path through multiple chips), the time spent while waiting for busy links to become idle, and the time for actual packet transmission at the given speed.

Fig. 4.8 on page 168 shows the model’s functioning in a particular example case: a sender at port$_1$ starts to send 2 packets to a receiver at port$_2$ at 0 $\mu$s and 3 $\mu$s in simulated time. Both packets are $m = 100$ bytes in size; they are transferred with 10 MByte/s and routed through 2 C104 chips, where each chip
Figure 4.5: a four-dimensional hypercube

Figure 4.6: a section of the GPMIMD machine
introduces a latency of $1 \mu s$.

The illustrations uses a graphical notation whose tokens are introduced in Fig. 4.7. The individual symbols represent the following semantics in left-to-right order: a solid line with a framed label indicates that an object (determined by the level on which the line starts) calls the \texttt{WAITFOR} method $y$ of another object $x$. A dashed line with a framed label indicates a call to a \texttt{TELL} method $y$ that starts execution after $n$ units of simulated time ($\mu s$ in this model). A solid line without a framed label indicates suspension of a coroutine for $n$ units of simulated time. Black and white circles denote that resources (semaphores) are granted or released.

For comparing the model against earlier published results from another simulation we chose an example where the text provided rich enough information for reproducing the required set of parameters. Random source produced packets at an overall production rate of $1$ MByte/s ($m = 32, \Delta t = 32 \mu s$) and sent them off to random destinations. Hypercube networks were set up with $16 \leq n \leq 512$, $l = \left\lfloor \frac{32-a}{d} \right\rfloor$, $a = 10$. Fig. 4.9 on page 169 shows that both models’ predictions for the resulting average packet delay (including transmission times) are in close correspondence. (The dark line refers to the present model’s predictions.)

Fig. 4.6 shows a representative part of the physical connections between C104 switches and T9000 transputers as present in a relevant GPMIMD machine configuration at CERN. The machine had 2 switch-boards with 4 C104s and 6 mother-cards with 5 C104s each. The switch-cards effectively formed the upper stage of a two-stage switching network.

Each T9000 transputer connected to 4 different C104 switches on the same motherboard with its 4 independent links (4 logical sub-networks). All connections between C104s were 2 physical links wide (group adaptive routing). 4 out of 5 C104s on each motherboard were set up identically with 3 double connections to C104s on each switch-card and to the fifth C104 on the same motherboard respectively.

For the reported tests only transputer links that connected to the first C104 per mother-card, and only 6 out of 7 T9000s per group were used. (In practical terms this meant that multitasking on transputers was less of an issue, because only 1 out of 4 physical links was operating. In Occam traffic on separate links is typically handled by dedicated processes, because of the language’s exclusively blocking communication semantics.)

![Figure 4.7: graphical notation for Fig. 4.8 on the following page](image-url)
Figure 4.8: an example of the C104 model’s functioning
Figure 4.9: comparison between the model’s and earlier results \( a = 10, m = 32 \)

This made for a total number of 36 T9000s (20 MHz versions). 18 (on 3 mother-cards) acted as senders and the other 18 (on the remaining mother-cards) acted as receivers. Results from modeling and earlier measurements were reported as showing good agreements [76].

169
Verehrtes Publikum, jetzt kein Verdruss:
Wir wissen wohl, das ist kein rechter Schluß.
Vorschwebte uns: die goldene Legende.
Unter der Hand nahm sie ein bernes Ende.
Wir stehen selbst enttäuscht und sehn betroffen
Den Vorhang zu und alle Fragen offen.
BERTOLT BRECHT [ :-) ]
Glossary of acronyms

Begriffe ohne Anschauung sind leer,
Anschauung ohne Begriffe ist blind.
Immanuel Kant

R U sure?
The early AIM-65 microcomputer used this condensed console prompt in order to save a few (4) bytes in its then precious ROM.

AAL ATM Adaptation Layer (ATM)
AFS Andrew File System
ANSI American National Standards Institute
API Application Programming Interface
ATLAS A Toroidal LHC ApparatuS (CERN)
ATM Asynchronous Transfer Mode
BSD Berkley Software Distribution
BWX Byte and Word manipulation eXtensions (Alpha)
CAD Computer-Aided Design
CASE Computer-Aided Software Engineering
CBR Constant Bit Rate (ATM)
CC-NUMA Cache Coherent Non-Uniform Memory Access
CERN European Laboratory for Particle Physics (Laboratoire européen de physique des particules)
CISC Complex Instruction Set Computer
CMS Compact Muon Solenoid (CERN)
COFF Common Object File Format
COMA Cache Only Memory Architecture
CORBA COnmmon Request Broker Architecture
CPI Clock cycles Per Instruction
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Check</td>
</tr>
<tr>
<td>CSR</td>
<td>Control and Status Register (SCI)</td>
</tr>
<tr>
<td>DAQ</td>
<td>Data AcQuisition</td>
</tr>
<tr>
<td>DCS</td>
<td>Detector Control System (ATLAS)</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DRAM</td>
<td>Dynamic Random Access Memory</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>DVSM</td>
<td>Distributed Virtual Shared Memory</td>
</tr>
<tr>
<td>ECP</td>
<td>Elan Communication Processor (Meiko)</td>
</tr>
<tr>
<td>FCT</td>
<td>Flow Control Token (C104)</td>
</tr>
<tr>
<td>FDDI</td>
<td>Fiber Distributed Data Interface</td>
</tr>
<tr>
<td>FEX</td>
<td>Feature Extraction (ATLAS)</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In, First Out</td>
</tr>
<tr>
<td>FLOPS</td>
<td>Floating-Point Operation Per Second</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>GNU</td>
<td>GNU is Not UNIX</td>
</tr>
<tr>
<td>GPMIMD</td>
<td>General Purpose, Multiple Instruction, Multiple Data (CERN)</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HEP</td>
<td>High Energy Physics</td>
</tr>
<tr>
<td>HIPPI</td>
<td>High-Performance Parallel Interface</td>
</tr>
<tr>
<td>I/F</td>
<td>Interface</td>
</tr>
<tr>
<td>IEEE</td>
<td>international Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IPC</td>
<td>Inter-Process Communication</td>
</tr>
<tr>
<td>ISO</td>
<td>International Standards Organization</td>
</tr>
<tr>
<td>JTAG</td>
<td>Joint Test Action Group</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>LED</td>
<td>Light Emitting Diode</td>
</tr>
<tr>
<td>LEP</td>
<td>Large Electron Positron collider (CERN)</td>
</tr>
<tr>
<td>LHC</td>
<td>Large Hadron Collider (CERN)</td>
</tr>
<tr>
<td>LSB</td>
<td>Least Significant Bit (or Byte)</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>LVI</td>
<td>Link VME Interface (ATLAS)</td>
</tr>
<tr>
<td>LVL</td>
<td>LeVeL (ATLAS)</td>
</tr>
<tr>
<td>MDT</td>
<td>Monitored Drift Tubes (ATLAS)</td>
</tr>
<tr>
<td>MIMD</td>
<td>Multiple Instruction streams, Multiple Data streams</td>
</tr>
<tr>
<td>MIPS</td>
<td>Million Instructions Per Second</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>MPP</td>
<td>Massively Parallel Processor</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bit (or Byte)</td>
</tr>
<tr>
<td>MSGC</td>
<td>Micro-Strip Gas Counters (ATLAS)</td>
</tr>
<tr>
<td>MUX</td>
<td>MUltipleXer</td>
</tr>
<tr>
<td>NDL</td>
<td>Network Description Language (transputer)</td>
</tr>
<tr>
<td>NFS</td>
<td>Network File System (UNIX)</td>
</tr>
<tr>
<td>NORMA</td>
<td>NO Remote Memory Access</td>
</tr>
<tr>
<td>NOW</td>
<td>Network Of Workstations</td>
</tr>
<tr>
<td>NUMA</td>
<td>Non-Uniform Memory Access</td>
</tr>
<tr>
<td>ODBMS</td>
<td>Object Oriented DataBase Management System</td>
</tr>
<tr>
<td>OSI</td>
<td>Open Systems Interconnect</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
</tr>
<tr>
<td>PCI</td>
<td>Peripheral Component Interconnect</td>
</tr>
<tr>
<td>POSIX</td>
<td>Portable Operating System Interface</td>
</tr>
<tr>
<td>PS</td>
<td>Proton Synchrotron (CERN)</td>
</tr>
<tr>
<td>PVC</td>
<td>Permanent Virtual Circuit (ATM)</td>
</tr>
<tr>
<td>PVM</td>
<td>Parallel Virtual Machines</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service (ATM)</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>ROB</td>
<td>ReadOut Buffer (ATLAS)</td>
</tr>
<tr>
<td>ROD</td>
<td>ReadOut Driver (ATLAS)</td>
</tr>
<tr>
<td>ROI</td>
<td>Region Of Interest (ATLAS)</td>
</tr>
<tr>
<td>ROM</td>
<td>Read-Only Memory</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
</tbody>
</table>

173
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RPC</strong></td>
<td>Resistive Plate Chambers (ATLAS)</td>
</tr>
<tr>
<td><strong>SAN</strong></td>
<td>System Area Network</td>
</tr>
<tr>
<td><strong>SCI</strong></td>
<td>Scalable Coherent Interface</td>
</tr>
<tr>
<td><strong>SCT</strong></td>
<td>Semi-Conductor Tracker (ATLAS)</td>
</tr>
<tr>
<td><strong>SFI</strong></td>
<td>Switch-to-Farm Interface (ATLAS)</td>
</tr>
<tr>
<td><strong>SHRIMP</strong></td>
<td>Scalable High-performance Really Inexpensive Multi-Processor</td>
</tr>
<tr>
<td><strong>SPARC</strong></td>
<td>Scalable Processor ARChitecture</td>
</tr>
<tr>
<td><strong>SPEC</strong></td>
<td>Standard Performance Evaluation Cooperative(^2)</td>
</tr>
<tr>
<td><strong>SPS</strong></td>
<td>Super Proton Synchrotron (CERN)</td>
</tr>
<tr>
<td><strong>SRAM</strong></td>
<td>Static Random Access Memory</td>
</tr>
<tr>
<td><strong>SVC</strong></td>
<td>Switched Virtual Circuit (ATM)</td>
</tr>
<tr>
<td><strong>SVR4</strong></td>
<td>UNIX System V Release 4</td>
</tr>
<tr>
<td><strong>TCP</strong></td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td><strong>TDM</strong></td>
<td>Time Division Multiplexing (ATM)</td>
</tr>
<tr>
<td><strong>TRD</strong></td>
<td>Transition Radiation Detector (ATLAS)</td>
</tr>
<tr>
<td><strong>TRT</strong></td>
<td>Transition Radiation Tracker (ATLAS)</td>
</tr>
<tr>
<td><strong>UBR</strong></td>
<td>Unspecified Bit Rate (ATM)</td>
</tr>
<tr>
<td><strong>UMA</strong></td>
<td>Uniform Memory Access</td>
</tr>
<tr>
<td><strong>UMP</strong></td>
<td>Universal Message Passing (MEMORY CHANNEL)</td>
</tr>
<tr>
<td><strong>UNIX</strong></td>
<td>UNiplexed Information and Computing System (UNICS)</td>
</tr>
<tr>
<td><strong>URL</strong></td>
<td>Universal Resource Locator</td>
</tr>
<tr>
<td><strong>VCI</strong></td>
<td>Virtual Channel Identifier (ATM)</td>
</tr>
<tr>
<td><strong>VHDL</strong></td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td><strong>VHSIC</strong></td>
<td>Very-High-Speed Integrated Circuit</td>
</tr>
<tr>
<td><strong>VI</strong></td>
<td>Virtual Interface</td>
</tr>
<tr>
<td><strong>VPI</strong></td>
<td>Virtual Path Identifier (ATM)</td>
</tr>
<tr>
<td><strong>WAN</strong></td>
<td>Wide Area Network</td>
</tr>
</tbody>
</table>

\(^2\)for comparison: \(1 \text{ SPECint}^{95} = 40 \text{ SPECint}^{92} = 10 \text{ CERN units} \approx 40 \text{ MIPS}\) [6]
Bibliography

Was zog Dich her, wenn nicht der Kunde Wunsch?
Kundry


3ftp://www.cern.ch/pub/Atlas/TP/PS/tp0.ps.Z
4http://atlasinfo.cern.ch/Atlas/GROUPS/SOFTWARE/TDR/TDR.bk.ps
5http://www.cs.princeton.edu/shrimp/papers/hotinterconnect94.ps

175


8http://www.ph.rhul.ac.uk/research/hep/C40.ps
11http://cmsinfo.cern.ch/cmsinfo/TP/TP.html


---

15http://www.ph.rhbnc.ac.uk/research/hep/LVI_Rev2.ps
20http://sunrans.cern.ch/rhauser/t2demo/msg.ps
[34] R. Hauser. *An Implementation of the SCI PHY-API (IEEEStd P1596.9 Draft 0.41) for Dolphin PCI–SCI Boards (Draft 1.2).* CERN, ATLAS internal note, 1997.


---

21http://www.cern.ch/rd11/subfarm/sci1.ps
23http://www-pnp.physics.ox.ac.uk/atlas/theses/rjh/rjhindex.html
25http://www1.cern.ch/Atlas/GROUPS/DAQTRIG/MODELLING/rt95/paper.ps


\(^{27}\)http://www.digital.com:80/info/hpc/ref/mess_pass.pdf

\(^{28}\)http://www-flash.stanford.edu/~ubetcha/papers/postscript/tpds.ps


---

31http://hepwww.rl.ac.uk/atlas/12/urd/urd_draft8.txt
33http://www.omg.org/corba2/corb2prf.html


---

36 http://www.cs.cornell.edu/Info/People/tve/thesis/
37 http://hepwww.rl.ac.uk/atlas/l2/t2data/defs.html
That's all folks.
Buggs Bunny

version 0.99a