Report from

ARTES Summerschool

Linköping, August 23-27, 1999

Report version: ARTES 1.0
Report printed: 30 August 1999

Jakob Engblom
Department of Computer Systems
Uppsala University
jakob@docs.uu.se

ARTES Summer School

This report includes summaries of all the invited talks and tutorials, as well as a short section on the ARTES & PAMP evaluations. In most cases, pictures of the speakers have been inserted.

Tutorial:

Design of Dependable Real-Time Systems in the Time-Triggered Architecture

Hermann Kopetz, Technische Universität Wien

Famous from the MARS project in Wien, and its spin-off technology: the Time-Triggered Architecture or Protocol (TTP). Purpose of lecture: give an idea of how to design dependable computer systems using a certain architecture

Kopetz is interested in hard real-time systems for dependable (also known as safety-critical) computing. A hard real-time system is *fundamentally different* from soft real-time systems. Hard real-time system is characterized by the following properties:

- Paced by the environment: the environment sets the deadlines, not the computer. The deadlines are not necessarily short, but note that they are still usually "hard". Also note that in some cases the man-machine interface is a part of the control loop (like fighter aircraft) and thus subject to hard deadlines.
- Predictable performance under worst-case loads, worst-case scenarios. Always design for the peak load scenario, not the normal load.
- Short time-spans: humans cannot intervene to do error detection (in soft real-time systems humans can interfere).
- We need continued service = not possible to use standby redundancy. Must use parallel redundancy, if at all.
- Data has a short time of validity (Kopetz considers the validity time of data to be one of the most important factors for real-time systems). If the data about the world is not up-to-date, wrong decisions might be made. Needs *temporal awareness* in the system. Data needs to be in some way tagged with its temporal validity (a time interval).
- The real-time computer system is *always* embedded in a larger system, containing an environment (controlled object) and an operator interface.

❶ *Time-Triggered Architecture*

Temporal Validity of Data

Temporal validity of data is *the key* to the real-time architecture (rather than scheduling). Scheduling is one technique to keep data valid: *temporal data validity* is the high-level driver. The communications subsystem in a distributed real-time system needs to be aware of the temporal constraints of data. It is not possible to just put data into a queue and send it when possible: we need to operate on temporally valid data.

RT Entity: something that generates real-time data. Outside its field of control, it cannot be changed, only observed and propagated. Examples: the desired outcome of the user, the value of some sensor in the real world, the result of a control algorithm in the computer system.

`Observation=<Name, Time, Value>.`

- `Name` = name of real-time entity.
- `Time` = the time of the observation. We need a common notion of time, i.e. a global clock. Physical time, not logical time.
- `Value` = the value of the entity.

For a distributed real-time system, *messages* carry *observations*. Both messages and observations are *atomic*: cannot be taken apart. And the time needs to be the same at all nodes: *global time* is absolutely crucial to real-time systems.

RT Image: temporally valid picture of the real-time entity. The picture must be periodically updated to maintain its validity, either by observing the entity or by estimating the state using a model of the environment[1]. We can trade: either frequent observations (high bandwidth, low computation) or infrequent observation with estimates (low bandwidth, high computation).

Membership

Very important property for distributed systems. All nodes must have a correct view of the status of other nodes, and all nodes must have the *same* view of the set of living nodes. And the status must be *timely*: the information must be propagated within some deadline.

Example: for an ABS braking system, all nodes must agree on which nodes are alive and working. Note that a node can be dead both because of computer failure and physical brake failure. The control loop needs to know which nodes are alive, since that allows it to brake the car correctly (you can brake a car with just three wheels braking, using a different force distribution).

Model of Time

For a distributed system, the *ticks* of the clocks are going to be *blurred* events: not all clocks will tick at exactly the same moment in real time. Solution: use a *temporal grid*, with a resolution of four times the required precision of the digital clock (minimum interarrival time of events). Using four points on the temporal grid, all events can be ordered without an agreement protocol. We now have a *discrete* time, providing reliable time stamps.

Conclusion: reliable time stamps are part of the TTP hardware.

Events or States?

Event-based view of the world: informs about the difference between an old state and a new state. Requires agreement on the state between the sender and the receiver. Lost messages can give serious problems with lost synchronization, duplicate messages, etc. Requires “precisely once” semantics.

State-based view of the world: focuses on the state of the system at a particular point in time. Requires larger messages than event messages. State messages are self-synchronizing: it does not matter much if a message is lost or duplicated. Enough with “at least once” semantics.

[1] Example: engine control, where we can (must) use a model of the engine to predict the state of the system at the time that the observation message reaches the controller (the bandwidth is too small and the change of the system is too high to allow raw observations to be valid at the time they reach the controller).

A Real-Time System is **Event Triggered** if the control signals are derived solely from the occurrence of events. A RT System is **Time Triggered** if the control signals are derived solely from the progression of a (global) notion of time.

State messages make the peak load easier to determine, since we are not at the mercy of the environment regarding the number of events occurring (the worst-case event estimates may be inaccurate). State messages are sent periodically (time-triggered).

Notes:
- Really "incremental" vs "absolute" information exchange (according to the audience).
- Bringing faults into the system exposes the inherent robustness of the state approach. Note that the absence of a time-triggered message can be considered as a message about the failure of a node on the network[2].
- Each sensor has enough intelligence to convert state and event information into each other. Example: button states that simply remember whether the button has been pushed or released, based on events.
- At peak load, state-based systems are easier to validate.

Note: most systems are somewhere in between, this is black and white, but the world is grey. TTP is based on the state-based/time-triggered approach.

Locus of Temporal Control

Time-triggered systems: the temporal control is located in the *communications system*. The schedule or message description list is used to determine the communications pattern, who sends what when. Regular, pre-planned system. Easier to validate, since the hosts and the communications system can be considered as independent entities.

Event-triggered systems: temporal control in the *host computers*. The hosts determine when events are sent, forcing all analysis and validation efforts to consider the entire system (very high complexity, usually).

Conclusion: time-triggered systems are a good way to control the complexity of the systems. Complexity is controlled at the interfaces. Subsystems can be tested at the borders of the subsystems.

TTA and TTP

"Gives the best basis for building dependable time-triggered distributed real-time systems". The key concept is validity of information, the focus is on the temporal properties of real-time data at the communications controller. TTA: the *architecture*, the *idea*. TTP: the *communications protocol*.

Two versions of TTP:
- TTP/A: cheap, light, master-slave protocol. Central clock at the master. For simple sensor networks. Cost as low as 50¢ per node.
- TTP/C: the full fault-tolerant protocol. Much more expensive, much more capable.

The communications network interface (CNI) sits between the hosts and the communications systems, and provide the service of a *temporal firewall*.

Key ideas of TTP:
- Integration of all the basic services (no layering in order to enhance performance).
- Use of a priori knowledge (system design) to make it more efficient and better at error detection.
- Continuous state agreement (time, membership, and progress of message schedule).
- Fail-silent nodes.

[2] Used in the ABS braking system implemented in a Mercedes E-Klasse using TTA.

TTP/C Hardware and Marketing Considerations

Hardware manufacture:

- There exists a silicon implementation containing about 400 000 gates. This chip, designed at the TU Wien (within a European research project) is now sold by the *TT Tech* spin-off company (`www.tttech.com`).
- *Motorola* has licensed the technology and will implement TTP controllers onboard chips like the PowerPC, M*Core, and 68HC11.
- *Siemens, TI, ARM, Toshiba* have expressed interest, especially for making chips for the automotive industry.

The hardware contains several oscillators: the *bus guardians* have an independent clock and checks that a node can only transmit in its allocated time frames. This provides the fail silent behavior.

The message overhead is very small: only 20 bits for up to 16 bytes of data sent. Naming is associated with time slots, which means that two different nodes need not know of each other's naming (message identifiers etc.), which makes integration simpler.

Furthermore:

- NASA is interested in TTP for use in space and aeronautical applications. The mass-market (automotive industry) drive of TTP promises to bring fault-tolerant communications hardware down in cost by a factor of about 1000 (compared to the solutions used in aerospace today).
- Stanford has worked on formally proving (parts of) the TTP/C protocol.
- A comparison to the CAN bus can be found on the TT Tech homepage. CAN is appropriate for other domains than TTP.
- TTP Forum: annual meeting of companies interested in TTP. Sponsored by the European Union.

❷ Design Using the TTA

Limiting factor of future real-time systems: *not* silicon cost (hardware becomes fantastically much smaller and faster at current feature shrink speed), but the *mental effort* required to build systems of a certain complexity. An architecture should be a *constraining framework* that makes it possible to build the systems we want. Enforce the adherence to interfaces. "Architecture Design is Interface Design".

Methodology: the systems integrator (automotive company, for example) defines interfaces towards the subsystems (value and temporal domain – easier with TTP), and integrates the subsystems delivered by subcontractors and project groups. The interfaces are specified by "which data when".

Two types of services:

- **Prior services**: provided by the various subsystems.
- **Emergent services**: services emerging after system integration, properties of the system as a whole.

Requirement: composability. If you put two independent systems together, the function of each one of them should not affect the others. The "prior services" should not be affected. To achieve this requires complete specification of interfaces, including the *temporal domain*. Precise interface specification makes integration time shorter: no risk for critical instants, etc.

TTA uses top-down design:

- First define the component interfaces (which data gets delivered when?).
- Then design the internals of each components.

Note that each node is an integration of software and hardware. Software in itself has no temporal properties.

Real-time software cannot be component-based unless the timing aspects are taken into account. Component reuse requires temporal specifications, but usually only the functional properties are specified. It is better to share *complete components* (i.e hardware and software together) than pure software components. "You should never try to abstract away from your hardware when doing hard real-time systems".

Flexibility or Dependability

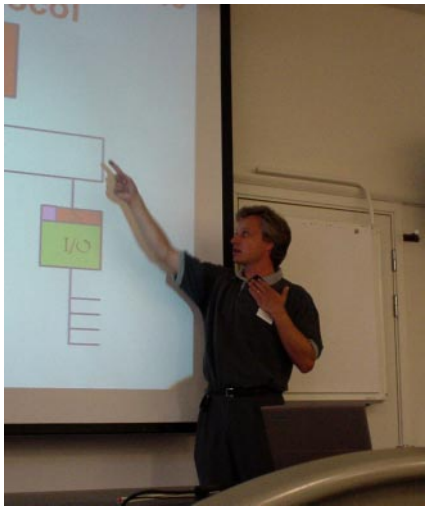
Kopetz: flexibility and dependability are *opposite* factors. When making things flexible, you give up information that can be used for error detection. As always, there are gray areas in-between.

Tutorial:

Parallel Architectures Today and Tomorrow

Erik Hagersten, Uppsala University (`eh@it.uu.se`*)*

Multiprocessing of Erik's interest: multiprocessors working on general problems. "Multiple CPUs working on the same problem". Nothing about real-time. Slides in the summer school file.



Shared-memory multiprocessors

Basic model: a shared memory, a number of processors each with a local cache, and an interconnect connecting all the components. Implicit replication of data to the caches of CPUs: several CPUs may have the same datum locally in their caches.

Cache Coherence

Coherence problem: some CPU changes the content of data that has been placed in other CPUs caches. Typical solution: **write-invalidation**: mark all copies of the datum as invalid (and the copy in main memory), using a broadcast. Next problem: how to find the most up-to-date copy (**coherent read**). Solved using broadcast too: ask who's got the most up-to-date datum (some CPU or). On write-back, the changed datum enters the shared memory.

Cache states:

- Invalid, shared, modified: basic requirements
- Exclusive, owned, etc…: allows more optimizations, but adds complexity.

Write update: every write updates all copies. Short latency, large bandwidth requirements. Only used once in a commercial system.

Read point-to-point: the memory has a catalog of who has the most up-to-date copy of a piece of memory. Reads proceed in two phases: obtain address of up-to-date owner, ask for it directly. Easier to scale to large systems (each CPU only sees the memory requests that actually involves it), can use a point-to-point communications network. Approximately doubles the cache-to-cache transfer latency, which is very bad for commercial workloads.

Memory reflection: at the time of data transfer in *read point-to-point*, the memory is updated. Saves latency for write-once/read-many objects. Avoids having the cache state "owner". Creates extra traffic for another type of sharing (migratory sharing).

Point-to-point invalidate: let the memory list the nodes that have copies of a certain datum. The memory sends out invalidate messages to the other affected nodes. Wait for acks from the other CPUs before updating the local value (otherwise you get memory ordering problems). Adds a ton of memory state, but reduces the number of CPUs that sees a write. Longer write latency due to acks.

How to implement state in memory?. At least a hundred different proposals published. One bit per CPU, linked pointers, lots of optimizations are possible. Example: the SCI (Scalable Coherent Interface) is built on a doubly-linked list!

The big arbiter of which type of system to create: time-to-market/complexity. You cannot create all systems you could imagine within the time given to produce a machine.

Memory Ordering

Ordering of accesses to multiple cache lines. If a CPU has observed that A' changes before B', what order should other processors observe? It depends!

- Contract between HW and SW guys. What can be expected? What must be implemented?
- The weaker the memory order, the faster the protocol.
  - Sequential consistency, SC [Lamport]: broadcast writes, all writes are sequential. Easy to use, but slow.
  - Total Store Order [Sindhu]. Implemented at SUN.
  - Processor Consistency [Goodman].
  - Weak Consistency [Dubois].
  - Release Consistency, RC [Gharachorloo]. Weakest model there is.
- Note that Speculative coherence actions can give SC almost the same performance as RC (according to current research).

Building Coherent Multiprocessors

There is no "right" architecture: the answer depends on what hardware, applications, etc. you have available and expect to run. Alternative architectures:

- **Single chip**: several CPU cores on one chip. All the way up to a four-way system ☺.
- **Single broadcast bus** interconnect: not very scalable (SUN Enterprise 6000, max 32 processors, Intel Xeon, max 8 processors).
- **Multiple broadcast busses** (SUN SC2000, CRAY CS6400): interleaved buses. Increased scalability, but ordering and pin count problems.
- **Data crossbar, address bus**: Separate the data into a separate bus, use broadcast for address queries but crossbar switch for data. Not implemented.
- **Data and address multicast crossbar**: Implement a broadcast using a hierarchical system behaving like a bus (electrically better than a long bus, allows 6 times the frequency). Sun Enterprise 10000. Increased bandwidth and scalability, but also more chips and latency.
- **Combined data and address point-to-point crossbar**: crossbar hardware implementing a point-to-point network. HP and Convex. Requires directory in memory, but gets great scalability.

General rule: more scalability implies:

- Higher latency
- More chips
- More non-uniformity
- Higher cost

Two price points for the interconnect:

- Sun E6500 (30-way): about 8% of the cost.
- Sun E10000 (64-way): about 16% of the cost. Most complex system.

UMA or NUMA

Uniform Memory Access (UMA): very high goal: same speed to all memory for all processors.

Non-Uniform Memory Access (NUMA): the speed of access to memory may differ in speed to different processors. Place a slice of the memory at each CPU node: very quick access, but much slower to other CPU's memory.

SUN Enterprise Server 6000 (E6000): Case Study of SMP

Basics:

- UMA system, despite the fact that memory is co-located with the CPUs. Reason: logical to build a system where memory is added together with CPUs (2 CPUs on one board).
- Decision: bad-news case is the common case: optimize the case when the data misses the current board. No shortcuts for accesses for local memory, which would make it a NUMA system.
- Broadcast backplane bus runs at 83-100 Mhz – and it cannot go much faster due to physics.

SUN made a number of different machines from the same basic hardware: all the way from E3000 up to E6000. 6-way to 30-way systems.

Erik showed some example boards from the E6000. Most of the volume: heat sinks!

Multiprocessor Economics

Multiprocessors: it's *not* all about scalability, it's about making money. Early 1990s: lots of companies. All of them are now gone – they misjudged the market: it was not lots of very fast CPUs… rather moderate number of moderate-speed CPUs (they all targeted the very small engineering high-performance computing segment). Scaling to 10 000s of CPUs simply was not needed very much.

Bulk of the money: commercial processing, below $250k machines. About the same time that all the massive parallelism companies went bust, multiprocessing really took off. With:

- Standard operating systems on standard CPUs
- Growing database size and the explosion of the Internet (and thus Internet servers)
- Ready-to-buy parallelized database software (no effort on part of the customer)
- Return of the central computing facility

HPC customers: very different type of customer. Reliability is not very important. Performance is. But they can only buy the same systems as used by the database commercial customers, i.e. <100 CPU SMPs.

But "scalability is fun"… only market left at the extreme high end: the **Accelerated Strategic Computing Initiative**, US Gov't. Needs 100 TF by 2004. Tactic: take technology from commercial database SMPs, and try to apply it to high-performance computing. Selected technology: cluster of SMPs (simpler to manage than a cluster of PCs). Two-knob approach: number of boxes 40-80, CPUs per box 16-128. Software is message-passing based and fault-tolerant.

Middle ground: 100-1000 CPUs. Wants it to feel like an SMP to software, but the hardware looks like NUMA. Solution: self-optimizing DSM (ex. SUN WildFire). More on that below…

SUN WildFire: a self-optimizing architecture

Built by Erik's group and the old Thinking Machines group in Boston. Scales to 112 CPUs, complementing and using SMP technology. Impose few changes on the applications and operating systems of the machines.

Approach: **self-optimizing Distributed Shared Memory (DSM).** 4 SUN E6000 servers combined into one cluster. Simple Cache-Only Memory Architecture (S-COMA coined at SiCS): using a combination of hardware and software to implement the memory architecture.

Memory architecture: The default mode is NUMA, and then OS decides which data to replicate (in COMA style). The result: the pages that most benefit the performance will be replicated (note that replication works per box: large nodes imply less overhead for replication). Specialized hardware to keep the nodes coherent – most complex chip ever developed at SUN. Processing: hierarchical affinity scheduler in OS (hacked Solaris 2.6): keeping data and processing co-located, if possible.

I have heard this talk once before, when Erik was "introducing" himself in Uppsala.

What Matters in Computer Architecture

- Physics rule! Cooling, pin-out (how many pins can you fit), connectors, …
- Reliability, availability, servicability, etc. is as important as performance.
- Testability and verifiability must be weighed against the performance of complex architectures. Complex to verify = longer time to market = hard to motivate. Requires 10-20 % performance increase to be worth it (per month of delay).
- For future expansion: wants one family of scalable building blocks. Late binding of how to configure it (number of submachines etc.).

Final Remark: Industry & Research!

Commercial architecture is about making money. *Researchers* must see beyond the making of money… but you need to have it mind if you want to influence the industry.

Tutorial:

Parallel Programming on Shared Memory Multiprocessors

Jaswinder Pal Singh, Princeton (Professor of Computer Science)

Per Stenström: "*The* authority on how to program parallel machines". PhD from Stanford.

Parallelizing examples:

- Ocean current simulation: very regular scientific computation. 2D-slices of the ocean: computer each slice in parallel, or compute all points on a slice in parallel.
- Galaxy Evolution: irregular scientific computation. Each star is calculated in parallel.
- Shear Warp Volume Rendering: render a set of points (millions of points in 3D) as an image.

Limits:

- Basic law of parallel computing: *Amdahl's Law*. If fraction *s* of computation is inherently serial, the speedup is limited by 1/*s*. Given only 5% essential sequentiality, the speedup is limited to 20.
- Note that many problems have a parallelism varying over time: might be quite serial at times, very parallel at others. Example: logic circuit simulation, where the number of firable gates can vary over time.

Process of Parallelizing Programs

Assume: we have a sequential algorithm. **Process**:

- Identify work that can be done in parallel

- Partition the work and maybe data among processes
- Manage data access, communications and synchronization.

Task: a unit of parallel processing (something done sequentially, undecomposed). How big to make the tasks? Fine-grained vs. Coarse-grained. *Process*: something that performs tasks (threads or processes or whatever). Task partitioned across processes. *Processor*: physical things that do things. Processes assigned to processors.

Steps:

- **Decomposition**: breaking an algorithm into tasks. Goal: find enough tasks to keep the processes busy, but not so many that communications become a problem. Can be performed as loop dependence analysis of a sequential *program*, but often better to go back to the fundamental *problem*: how can we solve it in parallel?[3]
- **Assignment**: specify a *mechanism* by which tasks can be given to processes. Balance the workload, reduce communication and management cost. Theory: NP-hard, in practice: a structured approach usually works well, not a big problem. Can be *static* or *dynamic*.
- **Orchestration**: How to: name data, structure communication (often determined by the hardware for implicit communication, as in SMPs. Harder for message-passing systems.), synchronization, temporal organization of data and tasks. The characteristics of the machine affect the decisions.
- **Mapping**: Architecture dependent; try to put processes on processors in such a way as to maximize the performance. Exploit locality in network, processors, etc.

Note: **Partitioning** = decomposition + assignment.

Programming for Performance

Very many (too many) trade-offs…

Parallel programming is successive refinement (the steps outlined above). Partitioning often independent of the machine, and then we try to handle interactions with architecture. View the machine as an extended memory hierarchy: try to reduce "artifactual" communications.

In some cases of the *same* problem, the best way to parallelize a problem depends on the relation between problem size and the number of processors. Because of communications effect – how much unnecessary data must be moved?

Load balancing for evolving problems: measure the work performed in previous time-step to determine the distribution of the work for the next time step. Works for galaxy simulation and graphics, among others. Very important technique. Typical method to partition work: orthogonal recursive decomposition (bisect the volume into equal-work halves repeatedly and recursively).

Try to organize the computation so that just one processor writes each piece of data. Multiple writers requires a lot of synchronization. Large read-only data sets are very nice for long-cache-line shared-memory machines.

Scaling to more processors may cause changes in the *partitioning*. Same thing goes for performance portability: when changing the underlying machine architecture, the programs often need to be repartitioning. However, performance scaling and performance portability have many problems and solutions in common: helps research.

[3] It is usually the case that a naive sequential program introduces unnecessary dependencies. The change of solution method may change the convergence properties of the computation.

Tutorial:

Distributed Asynchronous Real-Time Systems: Language, Middleware Support, and Application

Binoy Ravindran,Dept of Electrical and Computer Engineering, Virginia Tech

DARPA's QUORUM program (about 40 universities). Goal: infrastructure for emerging DoD applications in real-time field. Old-style real-time systems: centralized static regulatory control systems. New-styles:

- Inherently dynamic, distributed, require end-to-end adaptive resource management.
- Keywords: dynamic, distributed, QoS, adaption, …
- Direction towards COTS components. To make it easier to maintain the systems. Not so much reliability on the people who built and know the system (as is often the case for specialized military hardware).
- Systems evolve over time (example: Aegis cruiser)

Real-time properties of the systems:

- Required latencies and periods for functions like "radar scan → filter → target tracking". There is a *simple deadline* requirement: the time in which the complete computation must be finished.
- Some requirements are harder than others. The activation deadline (firing time) must be *known* and *hard*, since the time is used to predict the position of the target at the time the missile is launched.
- Problem: hard to predict the minimum interarrival time of the sporadic "shoot a missile" task. The Navy cannot provide any data on how often missiles will be launched.
- The missile guidance task (periodic) is triggered by a missile launch. At it requires using many of the same resources as the target tracking (radar, filter, etc.). It stops execution when the missile is done – the time of which is not known at launch time. Called *quasi-continuous*.

Goal of QUORUM: *Resource management middleware* that sits on top of hardware and operating systems and provides the services required by the real-time applications. Diagnose missed deadlines and reallocate resources to *provide acceptable timeliness*.

They use a *system description language* to configure the run-time system. Tells the run-time system what the desired *properties* of the system are (at quite a fine level of resolution: for each path through the system, we describe deadlines, allowed slack, how many invocations may miss their deadlines [e.g. 5/20]). Monitors the performance of the real-time applications, and performs appropriate actions to reallocate resources to keep meeting deadlines.

They have created quite a lot of mathematics to:

- Detect QoS violations: "*increased application latency*".
- Identifying recovery actions. Actions: migrate or replicate an application, depending on some diagnostics: are applications or hosts being overloaded?
- Resource allocation: where to put a migrated or replicated application? Localization and selection of resources. Both CPU and network resources taking into account.
- Make decisions from *path* violations to determine which *application* on the path to migrate or replicate.

For evaluation: a *benchmark* that approximates the Navy's real air-defense application. A large C++-application. Simulates the sensors and actuators. Visualization: looks like a video game ☺.

Research so far: create the middleware, create the benchmark for evaluation. Future research: Can the resource manager tell the Navy how many targets, missiles, etc. can be handled at a certain time. How to handle overload situations?

Research organization:
- Final deadline: construction of next-generation Aegis cruisers in 2003.
- The Navy sets goals for each year, the questions they want answered.
- Continuation, in a way, of the HiPer-D program[4].

Conclusions:
- A *path* is a convenient abstraction for expressing and enforcing QoS requirements.
- Middleware techniques are useful for delivering the desired QoS.
-

Tutorial:

Patent för Datorprogram

Ulf Kärn från Groth & Co, en patentbyrå. Föredraget hölls på svenska.

Patent = ensamrätt att tillverka, använda, eller sälja produkten. Patent kan vara beroende av andras patent, vilket kan orsaka att man själv inte kan utnyttja en uppfinning man själv gjort (och som man har patent på). Skyddstiden är 20 år från inlämningsdagen. Inom ett år måste man börja söka i andra länder, om man vill ha sitt skydd.

Ensamrätten gäller *inte*:
- Ej yrkesmässigt utnyttjande. Patentet skyddar det *kommersiella* utnyttjandet.
- Produkter som bringats i omsättning av patenthavaren. Dvs. när en produkt baserad på patentet har sålts till en kund kan man inte kräva mer pengar av kunden för utnyttjandet av patentet.
- Utnyttjande för experiment. Syftet med patentsystemet är att driva utvecklingen framåt.

[4] For more projects related to HiPer-D, see the "Case Studies" section of my RTAS '99 report.

Patentansökan:
- Bakgrund
- Uppfinningen
- Patentkrav: bestämmer vilket skydd som patentet ska ge. Kan innehålla ett stort antal olika mer eller mindre specifika krav. Man måste använda *allt* som står i patentet för att det ska vara ett intrång (en bil gör inte intrång på ett larvfotsförsedd fordon). Det gäller att vara så *generell* som möjligt i formuleringen. Balansgång mellan risken för att träffa något känt och värdet av skyddet. Patentkraven används för att sätta åt intrång. Ofta många olika sätt att beskriva samma sak, för att komma åt olika led av distributionen och produktionen av produkter, speciellt som lagarna är olika i olika länder.

Ej patenterbart:
- Upptäckt, vetenskaplig teori, matematisk metod, konstnärlig skapelse, datorprogram (gäller inte i USA), metod för affärsverksamhet eller spel, framläggande av information (dvs. användargränssnittet i datorprogram, lottokuponger, bingobrickor…).
- Notera att det gäller att formulera sig rätt när man skriver sin ansökan: detta gör det möjligt att patentera kryptoalgoritmer, metoder för programutveckling, etc.

För att bli patenterbart måste man uppfylla följande:
- **Nyhet**: Allt som skiljer sig från känd teknik. Känd teknik = allmänt tillgängligt för allmänheten, på *något* sätt. Om det har lämnat fabriken i någon form. *Kan* ha haft räcker. Mycket populärt sätt att knäcka patent.
- **Industriellt tillämpbar**: Måste lösa ett tekniskt problem (och det kan man *alltid* få till). Måste vara reproducerbart (får inte vara en slump, följer man instruktionerna ska det fungera).
- **Uppfinningshöjd**: Väsentligt nyskapande (stor fråga för diskussion mellan patentverk och sökande). Inte näraliggande för en fackman (dvs. inte en ”naturlig” lösning) – definition av fackman en viktig fråga för patent.

Patent kan ges för hela Europa via European Patent Office (EPO). En handläggning i Bryssel, men sedan måste det ändå översättas till respektive lands språk. Pris för att ta ett patent: 40-50 000 kr per patent och land för översättning och konsulter och behandling. Världspatent finns *inte*.

Datorprogram går ju inte att patentera… men:
- Europa: Datorprogram plus hårdvara → teknisk effekt. Det gör ingenting om hårdvaran inte är ny, bara effekten är ny. Styrsystem i en industriell process: sök på hela systemet. Olika sätt att lösa samma problem kan patenteras. Mjukvaran måste vara aktiv för att vara patenterbar.
- USA: Datorprogram, även när det ligger på diskett, kan patenteras. Skyddet gäller bara i USA. Europa på väg åt samma håll. Japan lika som USA.
- Sedan 1997: i USA 10 000 patent på datorprogram och relaterade saker, EPO 3 700.

Slutsats: datorprogramundantaget är på väg bort i Europa. Sverige har sagt sig följa EPOs praxis. EPO har godkänt två ansökningar på datorprogram, så antagligen kommer man snart kunna skydda datorprogram i Sverige.

USA: den som *uppfinner först* får patent, *inte första ansökan*. Gäller även om första uppfinning utanför USA. Man kan visa sin uppfinning i ett år innan man söker patentet (för experiment och försäljningssyften), så länge man inte säljer produkter baserade på patentet.

Tutorial:
Doktorer i Industrin

Anders Törne, Carlstedt Research (80 %) / Linköping universitet (20 %)

Budskap: Var man ska jobba beror vad man vill göra.
Erfarenhet:
- ABB Research 7 år
- Linköpings universitet 10 år
- Nu Carlstedt/LiU

Varför doktorer till industrin? Varför doktorera?
- Lära sig nyaste tekniken. Men det kan ju göra ändå.
- Doktorandutbildningen ger vana att självständigt hantera och lösa stora komplexa problem.
- Forskarutbildningen kan ge insikt i industriella problemställningar, med möjlighet till eftertanke och analys. Inte samma korta perspektiv, nedgrävda horisonter.

Behovet i industrin:
- Industrin behöver både spjutkompetens och bred förståelse.
- Industrin behöver människor som klarar att samarbeta. Inom universitetet kan man vara litet mer udda.
- Industrin behöver kompetensintegration (flera olika kompetenser i ett projekt, ska kunna prata med folk från många olika områden) och systematik. Ett systematiskt tillvägagångssätt är oerhört viktigt. Inget blint famlande (vilket är viktigt även inom forskningen, naturligtvis).
- Industrin behöver en affärsfokusering hos sina anställda. Tänk igenom de affärsmässiga aspekterna på forskningen – visa affärsnyttan med projekten.
- Det behövs både forskare och pragmatiker i industrin (kan naturligtvis kombineras i samma person).

Industri visavi akademin:
- Akademin: långsiktigt, upptäcktsorienterat, ibland långsamt och ofokuserat. Teori, små projekt, låt någon annan göra skitjobbet.
- Industrin: produktorienterat, kortsiktigt, deadlines, normalt sätt mycket fokuserat. Praktik, stora projekt, måste göra allt – även skitgörat. Och det handlar om att göra en vinst.

Förbereda sig för ”verkligheten”:
- Se till att vara med i praktiska projekt, implementering.
- Prata med industrin: det ska inte vara något problem att få kontakt.
- Ta reda på hur projekt fungerar i industrin. Den roliga delen är antagligen arkitekturfasen, inte realiseringen.

SNART'99 talk:

Engineering Computer-Based Systems

Harold (Bud) Lawson, Lawson Konsult AB. Active in computing since 1958.

Systems Engineering:
- “Systems Engineering”: specialty evolved after world war II.
- INCOSY: International Committee on Systems Engineering. Also changing in the direction.
- US, UK: special education programs for systems engineering.

Need for a new profession and discipline: **Computer-Based Systems Engineering**. Not just real-time, the whole content: from lifecycle issues to minute technical problems, how to control large projects, etc. 1990: IEEE founded a Technical Committee on Engineering Computer-Based Systems (TC-ECBS). Over 1300 members in 70 countries. Annual conference and workshop, as well as working groups. TC-ECBS have defined a curriculum for a Master’s in ECBS.

Message: Important for Sweden to get on the bandwagon and organize education in the field of ECBS. Both engineering programs and professional training courses.

Hermann Kopetz
and
Harold Lawson

The Critical Platform Issue

- Today's computer and communications systems contain significant *unnecessary complexity*
- Complexities permeate platforms of hardware and software components
- People use our complex computer systems to host many complex and critical applications.
- We are in the "**Black Hole of Complexity**".
- Please note that the issue is not limited to embedded systems: we use Pentium/Microsoft products to design and simulate our systems.

Today we have **"Busyware"**: lots of software that simply manages complexity in the platforms we use. We spend too little time designing "**Valueware**" and too much time designing middlewares etc. (the Busyware). What we want is to have a small simple stable platform: "**Stableware**". Change will come only after a few really bad disasters. A few has started to happen: the *Yorktown* disaster, banking systems failures, etc.

"IBM System/360 was a terrible step backward" – revolution in hardware, but introduced lots of new complexity. Very complex instruction set ill-suited to software development. The big mess resulting: OS/360. And the trend has continued into the x86 days. Terrible machines to build system software for. The MIPS ideas of simplified instruction sets are brilliant, but still even RISC CPUs are too complex today.

The instruction set *does* give us problems. It might cause the compiler to be more complex, introduces hard choices. Also note that operating systems need to handle the low-level yucky stuff. We build stacks of complexity above the hardware, which is fine until a disaster/bug happens, when debugging gets very difficult.

Examples of good architecture:
- The AXE design with forced separation between tasks.
- The SJ ATC system: the kernel has been stable since the 1970s (!)

Note: John Hennessy, *IEEE Computer August 1999*: Over the last 25 years, only performance has been emphasized in hardware development. Result: incredible complexity. Time for a change in computer architecture. .

There should be a way to obtain the same performance as today's systems using simpler hardware and software.

Invited Talk:

The Time-Triggered Model of Computation

Hermann Kopetz, Technische Universität Wien

Bridging the implementation gap: easier if the *specification* and *implementation* use the same basic concepts. *Implementation bias* in the specification.

- Computer Scientists have a tendency to want implementation-independent specifications. Not very valid in real-time systems, where resources are very important.
- *A design is only valid if you have an idea of how to implement it*[5].
- But note that the specification is more general and abstract than the implementation, even if they both use the same basic set of concepts.
- The Client-Server model (as employed by CORBA) is very poor model for real-time systems.
- The Time-Triggered model is a very good model for real-time systems.

The concepts used for modeling (and implementation) should support the problem domain (RT):

- Deadlines.
- Time-bounded validity of data.
- Distributed systems (all RT systems will be distributed in the future: intelligent sensors, etc.).
- Physical time is a first-class concept: we cannot do without time in the RT application domain. A specification ignoring time is not valid for a real-time system.
- State messages (see page 3).

Interfaces

Most important modeling concept: the *interface*. A unidirectional (both control and data) link from the sender node to the receiver node(s), maybe across the communications network. Precisely specified in time and value domains.

There is *no room for interrupts*, since control signals cannot propagate outside a node. Only data. Only way to find out what is happening: sample data frequently. This might be too restrictive for certain applications, but note that the implementation might use interrupts, although they are not part of the modeling language.

Client-Server Model for Real-Time???

Originally intended for soft real-time. Now: RT-CORBA: add time as an afterthought. Kopetz: skeptical about whether it is possible to retrofit time into a model. It is a too fundamental concept. CORBA tries to hide the OS, HW, Comms interface and still reason about the time. TTP: reason about time at the communications interface, below all the "Busyware" – where it is possible. Furthermore, there are no temporal firewalls in RT-CORBA, leading to no composability.

Message: using the time-triggered model we can narrow the implementation gap, making real-time systems easier to generate.

PanelEvaluation:

Evaluation of ARTES and PAMP

Purpose of ARTES and PAMP projects: to improve the quality of the projects by feedback from outside evaluators (Hermann Kopetz, Jaswinder Pal Singh, Lars Liljegren[6]).

ARTES projects:

- 17 graduate students funded in 14 projects. Financed for 2+2 years.
- Projects share a vision, but are not very closely coupled.
- Finances named specific graduate students for specific projects.

[5] Comparison: if you build a house, you allocate a budget and expect the architect to meet your cost objective.
[6] From ABB Automation Products.

PAMP: Symmetric Multiprocessors in High-Performance Real-Time Applications.

- 8 graduate students in 4 projects.
- "Closely-Coupled Multiprocessor": the projects are closely related.
- **Goal**: Design methods to effectively design multiprocessor-based applications.
- Application areas: multimedia clients and servers, information servers. Time-sensitive (QoS) applications with high computation and communications (bandwidth) needs.

Comments from the evaluators:

- The project is a little scattered: usually one or two PhDs per project. PAMP is a little better than ARTES. Better: set a grand target, let many people work on the problem(s). Find a great goal to work towards.
- Industry support: little *close* cooperation with industry. Better: maybe mandate internships (6 months?) for PhDs. Makes researchers more aware of industry problems. Let industry assign some resources.
- Target platforms: commodity systems of *today*. What about the systems of tomorrow? Should try to stay on the technology development curve. Little work related to I/O issues, which would be helpful.
- Create: demonstrators, larger projects, focus.
- Project methodologies: for large or small project groups.

The reviewers…