Chapter 6-2: Multiprocessor Software

Soo-Ik Chae

Topics

- Multiprocessor scheduling.
- Middleware and software services.
- Design verification.

6.3.3 Scheduling with dynamic tasks

- Can't guarantee that all tasks can be handled.
  - Can't guarantee start time for a process.
  - Unless the source of the tasks limits itself
- In a real time system, once we start a process, we want to guarantee its completion time.
- Admission control determines what processes can execute based on resources, load.
- For dynamic systems with more than one processors and/or tasks that have exclusion constraints, an optimal solution does not exists
  - Use heuristics

Ramarithram et al. myopic scheduling

- Assumptions:
  - Tasks are nonperiodic.
  - Tasks are executed non-preemptively.
  - No data dependencies between tasks.
- Task characterized by
  - arrival time: Ta
  - Deadline: Td
  - worst-case processing time: Tp
  - resource requirements {Tr}
- Original heuristic algorithm: $O(n^2)$
  - At each step, it checks whether the current partial schedule is strongly-feasible and if so, it applies the H heuristics function to all the remaining tasks.

Myopic scheduling algorithm

- Partial schedule is strongly feasible if the schedule itself is feasible and every possible next choice for a task also gives a feasible schedule.
- O(nk) version: myopic (short-sighted) scheduling algorithm
  - Constructs partial schedules.
    - Check strongly feasibility by evaluating an H (search metric) function
      - Shortest deadline first, shortest processing time, ..
    - Search includes backtracking.
  - Add a task to a partial schedule.
- Searches only first k tasks of the remaining tasks pre-sorted by deadlines.

Myopic scheduling algorithm

- {tasks_remaining}: arranged in the order of increasing deadlines
- Nr: # of tasks in {tasks_remaining}
- K: maximum number of tasks in tasks_remaining considered by myopic algorithm
- Nk: actual number oftasks in tasks_remaining considered by the myopic algorithm at each step of scheuling
  - Nk = min (k,Nr)
- {tasks_considered}: the first Nk tasks in {taks_remaining}

Load balancing

- It is a form of dynamic task allocation
- Move tasks to new processing element during execution.
- Task migration moves an executing task:
  - Homogeneous MP with shared memory: Just move the task's activation record from the old PE to the new PE
  - Heterogeneous MP with shared memory:Two versions of code should be available in both old and new PEs. Harder on heterogeneous multiprocessor.
- MP with non-shared memory
  - Need to copy all program data. Harder still if memory is not shared.

Load balancing scheduling

- Shin and Chang: schedule using a buddy list for each processing element.
  - List of other processing elements with which it can share tasks.
  - Subdivided into preferred list, ordered by communication distance to the buddy.
- When moving a job, search the buddy list in order, checking load until a satisfactory node is found.

Load balancing scheduling

Abstract—If task arrivals are not uniformly distributed over the nodes in a distributed real-time system, some nodes may become overloaded while others are underloaded. Consequently, some tasks cannot be completed before their deadlines, even if the overall system has the capacity to meet all deadlines. Load sharing (LS) is one way to alleviate this problem.

In this paper, we propose a decentralized, dynamic LS method for a distributed real-time system. Whenever the state of a node changes from underloaded to fully-loaded and vice versa, the node broadcasts this change to a set of nodes, called a *buddy set*, in the system. An overloaded node can select, without probing other nodes, the first available node from its *preferred list*, an ordered set of nodes in its buddy set. Preferred lists are so constructed that the probability of more than one overloaded node "dumping" their loads on a single underloaded node may be made very small.

6.4 Middleware and software services

- Operating systems provide services for shared resources in uniprocessors.
- Must generalize this notion for multiprocessors.
  - Need distributed information about resource state.
- Middleware provides services in distributed systems.
  - Generic services such as data transport.
  - Application-specific services such as signal processing.

Uses of middleware

- Middleware: coined for general-purpose systems to describe software that provides services for applications in distributed systems and multiprocessors.
  - It is not the application for itself, nor does it describe primitive services provided by the operating system.
  - Provides fairly generic data services such as data transport among the processor that may have different endianness or other data formatting issues.
  - It can also provide application-specific services

Purposes of middleware

- Services allow applications to be developed more quickly.
  - Which may be tied to a particular PE or an I/O device.
  - Alternatively, they may provide high-level communication services.
- Simplifies porting application to a new platform.
  - Middleware standards are particularly useful
- Ensures that key functions are correct and efficient.
  - Rather than rely on users to directly implement all functions, a vendor may provide middleware that showcases the features of the platform.

Middleware vs. libraries

- Traditional software libraries may provide functions but don't manage resources like OS.
- Middleware need to know global state, have privileges to manage resources by giving request to the OS on each processor to implement those decisions.
- Resources must be managed dynamically when requests come in dynamically.
  - Statically designing the system for worst-case costs too much.

Embedded vs. general-purpose middleware

- Embedded middleware must be very efficient:
  - Small software footprint.
  - Low latency.
  - Predictable performance.
- Embedded middleware may reside entirely within a chip or may communicate with other systems-on-chips.
- Middleware makes use of general standards:
  - Internet protocol (IP): often used
  - CORBA: widely used for distributed embedded services

CORBA

- Common Object Request Broker Architecture is widely used in business-oriented software.
- It is not a specific protocol
  - Rather meta-model using an object-oriented services.
- CORBA services are provided by objects that combine functional interfaces as well as data.
- An interface to an object is defined in an interactive data language (IDL)
  - It is language independent
  - Can be implemented in any programming language.
- Objects and their variables are typed.

What is the purpose / goals of CORBA?

- Enable the building of plug and play component software environment
- Enable the development of portable, object oriented, interoperable code that is hardware, operating system, network, and programming language independent
- How to meet the goals
  - Interface Definition Language (IDL)
  - Object Request Broker (ORB)

Interface Definition Language (IDL)

- Language Independence
- Defines Object Interfaces
- Hides underlying object implementation
- Language mappings exist for C, C++, Java, Cobol, Smalltalk, and Ada

Interface Definition Language (IDL)

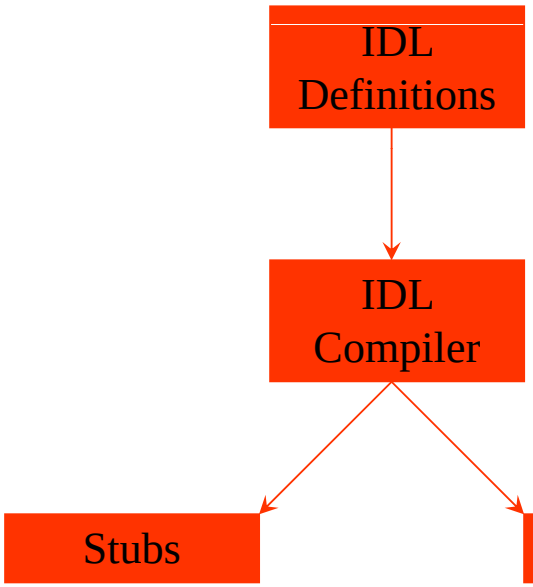
```
module <identifier>
   {
   interface <identifier> [:inheritance]
         {
         <type declarations>;
         <constant declarations>;
         <exception declarations>;
         <attribute declarations>;

        [<op_type>] <identifier>(<parameters>)
         [raises exception][context];
         }
   }
```

Defines a container (namespace)

Defines a CORBA object

Defines a method

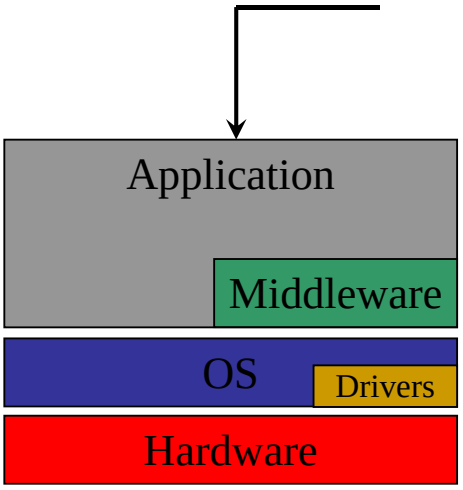
IDL Compiler



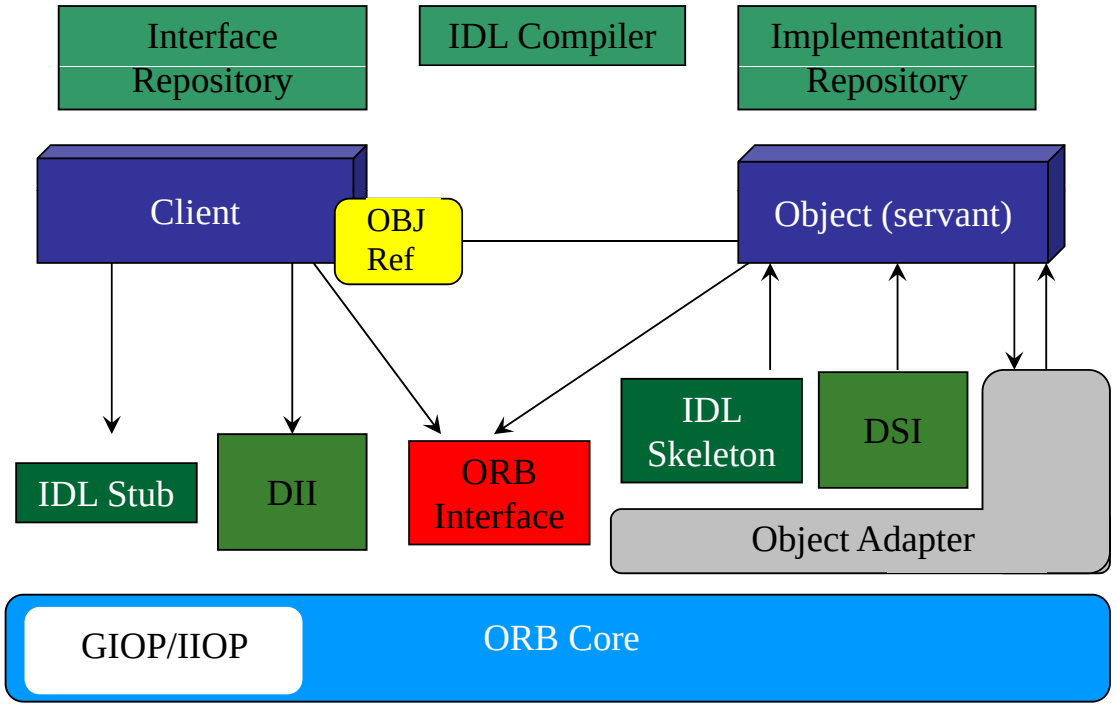
1. Define objects using IDL
2. Run IDL file through IDL compiler
3. Compiler uses language mappings to generate programming language specific stubs and skeletons

What is ORB?

- Implementation of CORBA specification
- Middleware product
- Conceptual Software Bus
- Hides location and implementation details about objects

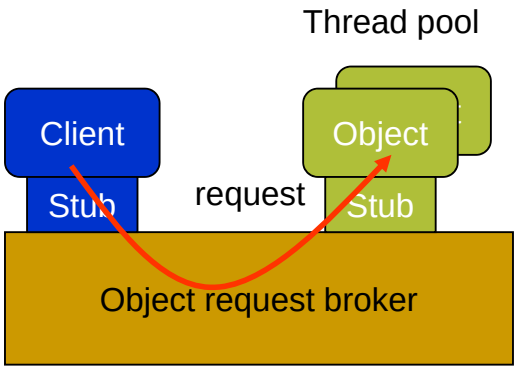


ORB Architecture



CORBA requests

- Requests handled by object request broker (ORB).
- Client and object may be on different machines.
  - ORBs may communicate.
  - A request to a remote machine can invkoe multiple ORBs
- The stub on the client-side provides the interface for the client while the skeleton is the interface to the object.
- A given service appears as an object but may be implemented with a thread pool.
- Each object instance has a unique object reference



ORB-to-ORB communication

- Provides mechanism for transparently communicating client requests to target object implementations
- Makes client requests appear to be local procedure calls
- GIOP – General Inter-ORB Protocol
- IIOP – Internet Inter-ORB Protocol

- The client's ORB and object's ORB must agree on a common protocol : IIOP

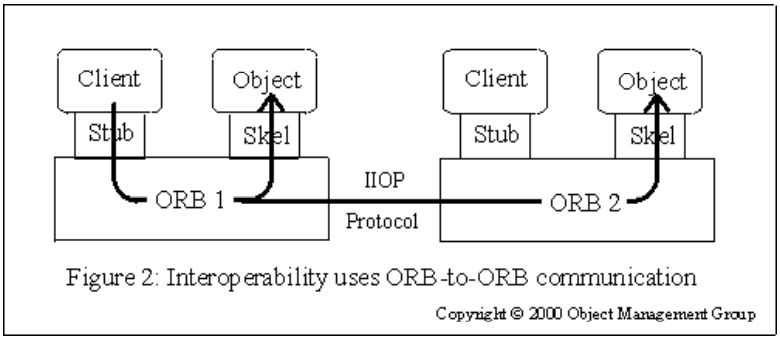
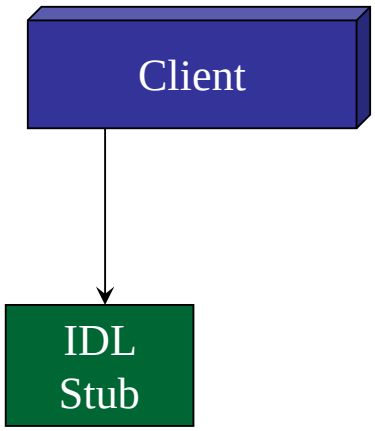


Figure 2: Interoperability uses ORB-to-ORB communication

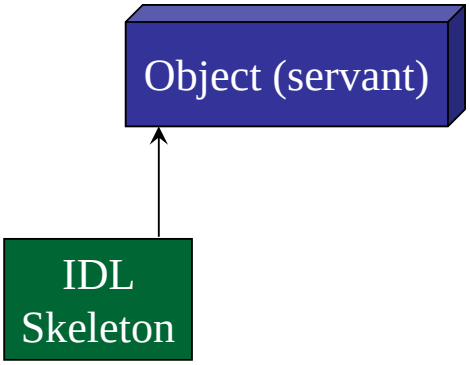
IDL Stub



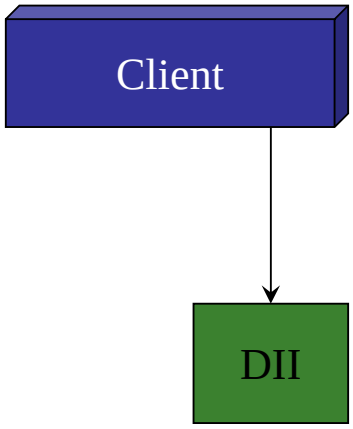
- Static invocation interface (SII)
- Marshals (encode) application data into a common packet-level representation
  - Network byte order (little-endian or big-endian)
  - Size of data types

IDL Skeleton

- Demarshals (decode) the packet-level representation back into typed data that is meaningful to an application
  - Network byte order (little-endian or big-endian)
  - Size of data types



Dynamic Invocation Interface



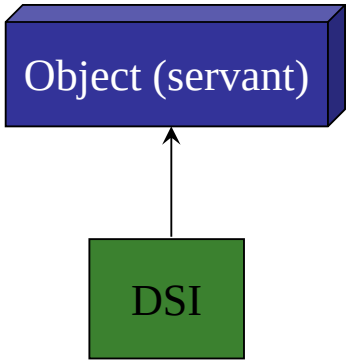
- Dynamically issue requests to objects without requiring IDL stubs to be linked in
- Clients discover interfaces at run-time and learn how to call them

Steps:

1. Obtain interface name
2. Obtain method description (from interface repository)
3. Create argument list
4. Create request
5. Invoke request

Dynamic Skeleton Interface

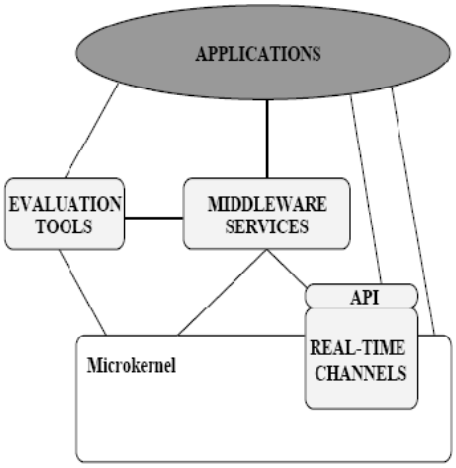
- Server side analogue to DII
- Allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of object it is implementing



ARMADA

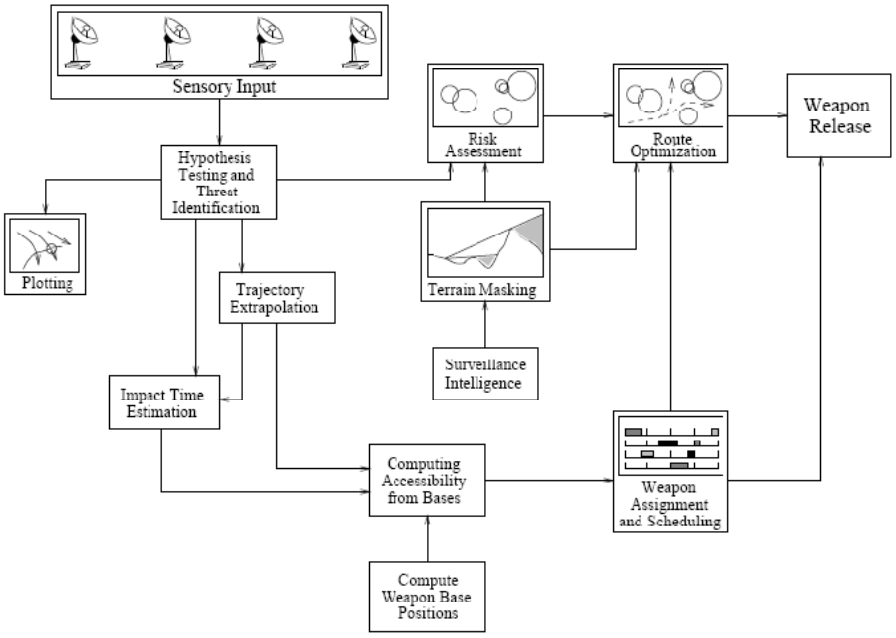
- Motivated by the requirements of large embedded application such as command and control, automated flight, shipboard computing, and radar data processing.
- Traditionally constructed from special-purpose hardware and software.
- A recent trend: build embedded systems using commercial-over-the-shelf (COTS) components

ARMADA

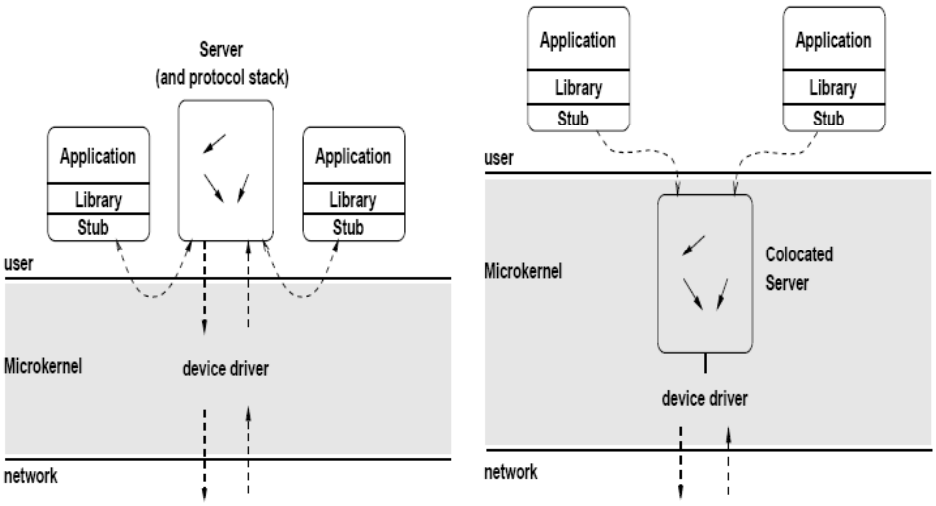


- Middleware system for fault tolerance and QoS.
  - Low-level real-time communication support
  - Middleware for group communication and fault tolerance.
  - Dependability evaluation and validation tools

A command and control application



General service implementation approach



(a) User-level server configuration (b) Co-located server

Figure 3. Service implementation.

General service implementation approach

The microkernel has to support kernel threads. The priority of threads executing in kernel space is, by default, higher than that of threads executing in user space. As a result, threads run in a much more predictable manner, and the service does not get starved under overload. Furthermore, the in-kernel implementation of x-kernel on our platform replaces some of the threads in the device driver by code running in interrupt context. This feature reduces communication latencies and makes the server less preemptable when migrated into the microkernel. However, since code executing in interrupt context is kept to a minimum, the reduction in preeptability has not been a concern in our experiences with co-located code.

Figure 3-a and 3-b illustrate the configurations of user-level servers and co-located servers respectively. An example of server migration into the kernel is given in the context of the RTCAST service in Section 4. The RTCAST server was developed in user space (as in Figure 3-a), then reconfigured to be integrated the into the kernel (as in Figure 3-b). Whether the server runs in user space or is co-located in the microkernel, client processes use the same service API to communicate with it. If the service is co-located in the kernel, an extra context switch to/from a user-level server process is saved. Automatically-generated stubs interface the user library (implementing the service API) to the microkernel or the server process. These stubs hide the details of the kernel's local communication mechanism from the programmer of the real-time service, thus making service code independent from specifics of the underlying microkernel.

Real-time communication service architecture

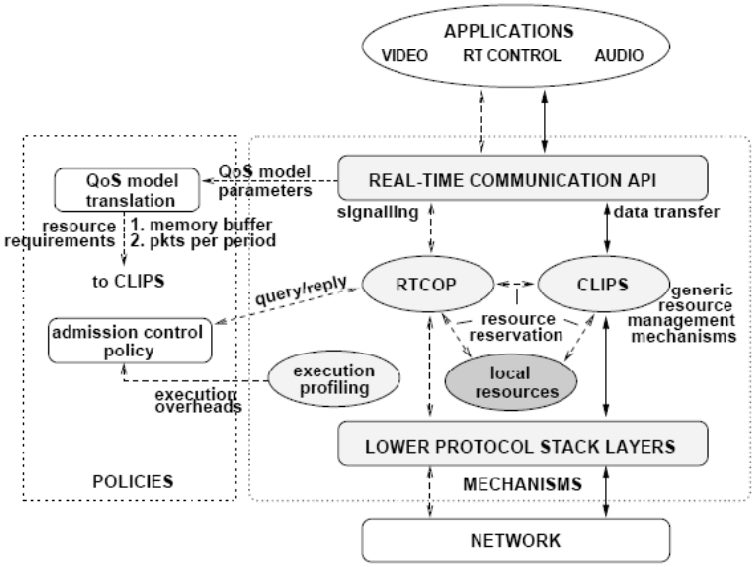


Figure 4. **Real-time communication service architecture**: Our implementation consists of four primary architectural components: an application programming interface (RTC API), a signaling and resource reservation protocol (RTCOP), support for resource management and run-time data transfer (CLIPS), and execution profiling support. Dashed lines indicate interactions on the control path while the data path is denoted by the solid lines.

Real-time communication service architecture

- Architectural requirements for QoS
  - Performance isolation between connections such that errors in one channel does not starve another channel.
  - Service differentiation (priority)
  - Graceful degradation in the presence of overload.
- CLIPS: a communication library for implementing priority semantics
  - Provide resource management mechanism
  - A clip is an object that guarantees a certain throughput in terms of the number of packets sent via it per period. And implements a configurable buffer to accommodate bursty sources.

Real-time communication service architecture

- RTCOP: real-time connection ordination protocol
  - Manages requests to create and destroy connections.
  - A clip is created for each end of the channel.
  - Each clip includes a message queue at the interface to the objects, a communication handler that schedules operations, and a packet queue at the interface to the channel.
  - The communication handler is scheduled is scheduled using an EDF policy.

MPI (multiprocessor interface)

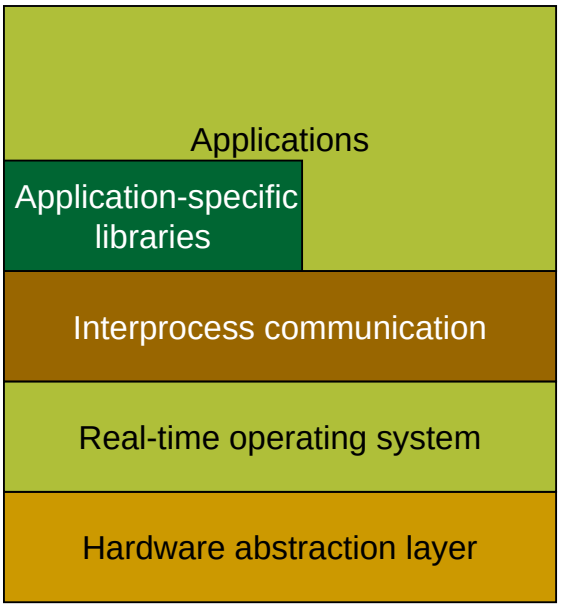
- A specification for middleware interface for multiprocessor communication.
- Widely used in scientific clusters.
- Decouples architectural parameters (# PEs) from algorithmic parameters (# data elements).
- Six basic MPI functions:
  - MPI_Init().
  - MPI_Comm_rank(): get the index of this node
  - MPI_Comm_size(): get the total number of nodes
  - MPI_Send().
  - MPI_Recv().
  - MPI_Finalize(): clean up
- The basic MPI communication functions (MPI_send(), MPI_recv())
  - Point-to-point, blocking communication

Software stacks in MPSoCs

- MPSoC resulted in a new generation of custom middleware that relies less on standard services and models.
- SoC middleware has been designed from scratch for several reasons.
  - Often power or energy constrained. Any services must be implemented very efficiently.
  - Although SOC may be committed with outside standard services, they are not constrained to use standards within the chip.
  - Today's SoC are composed of a relatively small number of processors. (simple yet)
- Software stack manages resources, abstracts hardware details.
- Performance, power requirements dictate a shorter stack than in general-purpose systems.

Typical MPSoC stack

- Application layer provides user function.
- Application-specific libraries are tailored to provide utilities for computation or communication specific to the application
- Interprocess communication provides services across multiprocessor.
- RTOS controls basic system functions.
- HAL uniformly abstracts basic hardware services.



Multiflex programming env (Paulin)

- Paulin et al.: uses hardware accelerators plus software to provide multiprocessor communication.
- Supports two parallel programming models:
  - DSOC: Distributed system object component
  - SMP: Symmetric multiprocessing
- DSOC is an object-oriented model.
  - It is a message-passing model and it supports a very simple CORBA-like interface definition (dubbed SIDL)
  - Client marshals data for call.
  - Server side unmarshals data for use.
- SMP engine uses memory-mapped reads/writes.
  - Supports concurrent threads accessing shared memory
  - The implementation performs scheduling, and includes support for threads, monitors, conditions and semaphores.

MultiFlex platform

Figure 1 depicts the *StepNP* flexible multi-processor SoC architecture platform, which was described in detail in [7]. The StepNP platform includes models of standard or configurable processors, a network-on-chip, configurable H/W processing elements, as well as networking-oriented I/O's. Aside from these domain-specific I/O's, this is a general-purpose platform.

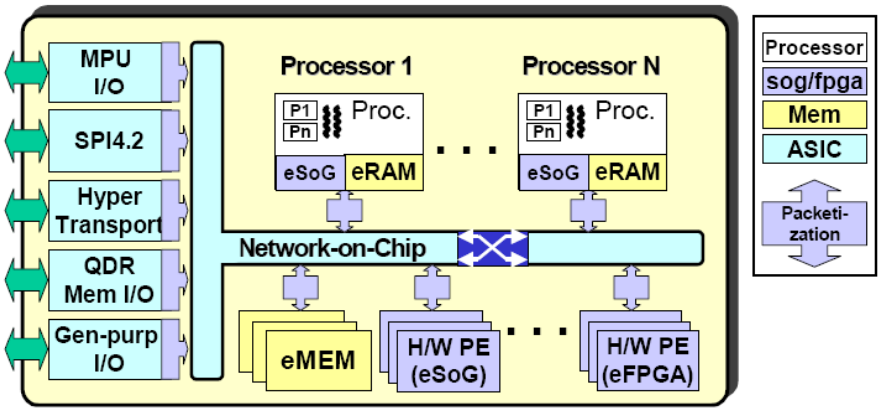


Figure 1. *StepNP* MP-SoC Platform

MultiFlex platform

Our implementation of the DSOC programming model relies on two key services. As we are targeting this platform at high performance applications, a key design choice is the implementation of these services in hardware.

- The hardware *Message Passing Engine* (MPE) is used to optimize inter-process communication. It translates outgoing messages into a portable representation, formats them for transmission on the network-on-chip, and provides the reverse function on the receiving end.
- The hardware *Object Request Broker* (ORB) engine is used to coordinate object communication. As the name suggests, the ORB is responsible for brokering transactions between clients and servers.

MultiFlex platform mapping

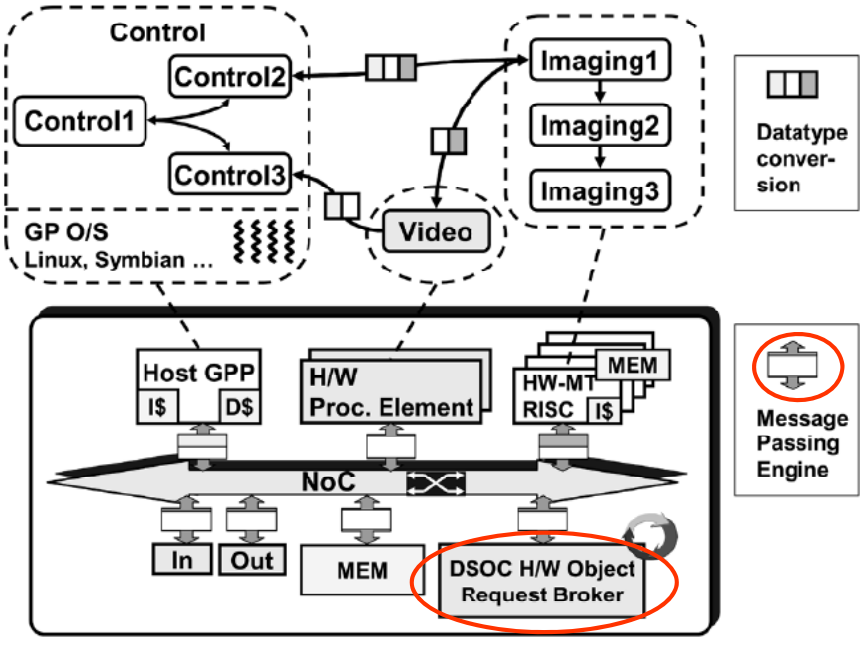
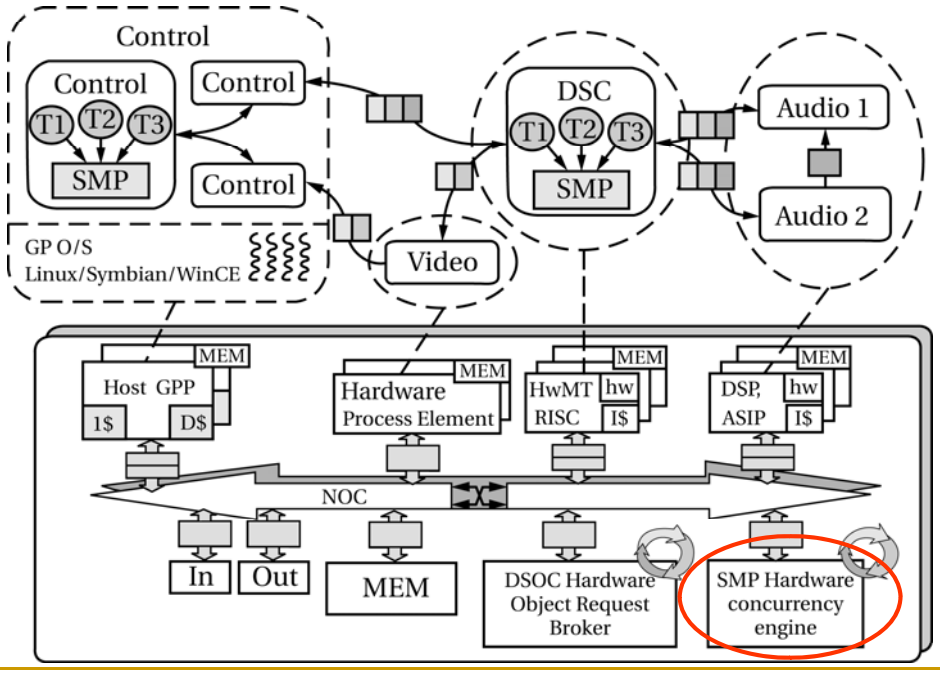


Fig. 2. DSOC model to platform mapping.

MultiFlex concurrency engine



Hardware concurrency engine

SMP functionality in the MultiFlex system is implemented by a combination of a lightweight software layer and a hardware *Concurrency Engine* (CE). The SMP access functions to the concurrency engine are provided by a C++ API. It defines classes and methods for threads, monitors (with enter/exit methods), condition variables (with methods for signal and wait), etc.

The CE appears to the processors as a memory-mapped device, which controls a number of concurrency objects. For example, a special address range in the concurrency engine could correspond to a monitor, and operations on the monitor are achieved by reading and writing addresses within this address range. Most operations associated with hundreds or thousands of instructions on a conventional SMP operating system are accomplished by a single read or write operation to a location in the concurrency engine.

Hardware concurrency engine

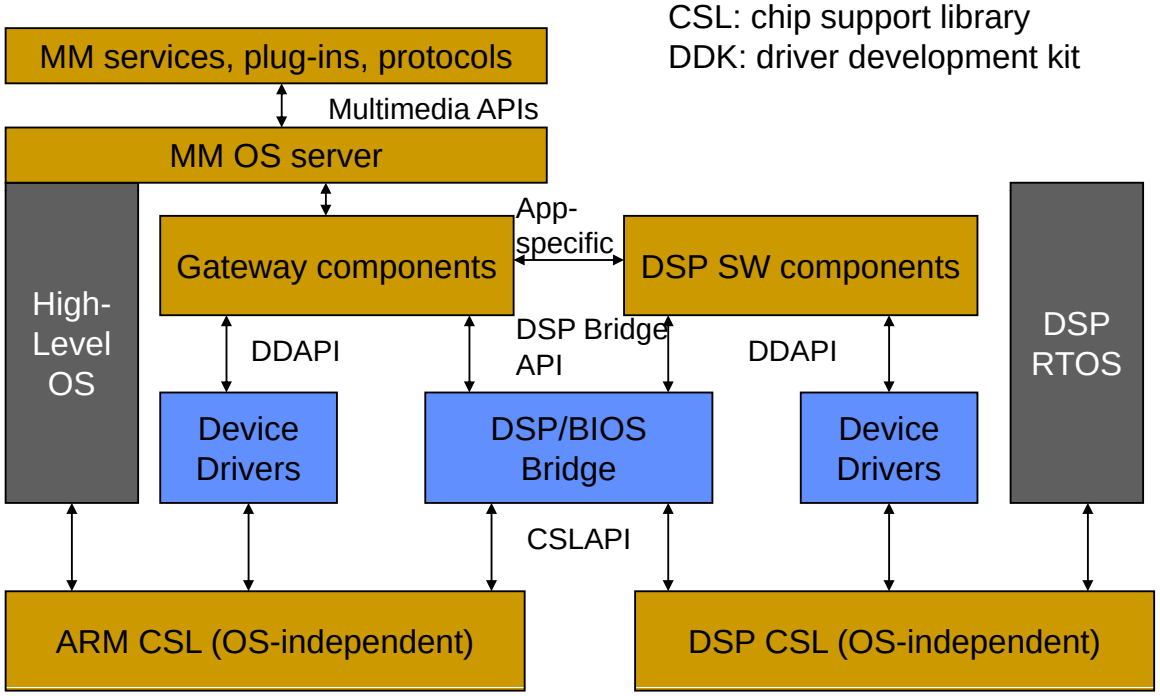
To motivate the need for a hardware concurrency engine, consider the traditional algorithm for entering a monitor. This usually consists of the following:

1. Acquire lock for monitor control data structures. This is traditionally done with some sort of atomic test and set instruction, with a spin and back-off mechanism for heavily contested locks.
2. Look at busy flag of monitor. If clear, the thread can enter the monitor. If the busy flag is set, the thread must: a) link itself into a list of threads trying to enter the monitor, b) release the lock for the monitor, c) save the state of the calling thread (e.g., CPU registers) and switch to another thread.

Monitor

Example: OMAP software platform

CSL: chip support library
DDK: driver development kit



OMAP software architecture

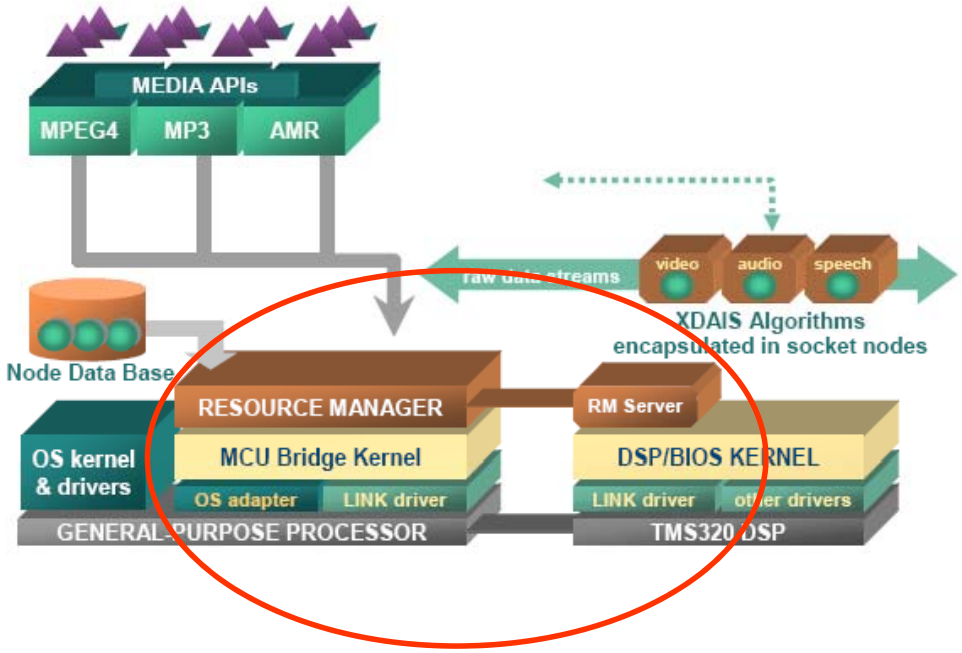


Figure 5-5. OMAP multi-processor software architecture

DSPBridge

- Abstracts the DSP software architecture for the general-purpose software environment.
- APIs include driver interfaces and application interfaces:
  - Initiate and control DSP tasks.
  - Exchange messages with DSP.
  - Stream data to/from DSP.
  - Check status.

Resource manager

- Provide API interface to the DSP.
  - Loads, initiates, and controls DSP applications.
- Keeps track of resources:
  - CPU time, memory pool, utilization, etc.
- Controls:
  - Tasks.
  - Data streams between DSP and CPU.
  - Memory allocation.

Multimedia messaging service

- Minimum requirement from spec:
  - JPEG, MIME text with SMS, GSM AMR, H.263, SVG for graphics.
- Optional: AAC, MP3, MIDI, MP4, and GIF.
- Must provide: MM presentation, user notification, MM message retrieval.
- Additional functions: MM composition, MM submission, MM message storage, encryption/decryption, user profile management.

Quality-of-service

- QoS must be measured system-wide.
  - One component can destroy system QoS characteristics.
- QoS modeling:
  - Contract specifies resources that will be provided.
  - Protocol manages the contract.
  - Scheduler implements the contract.
- Resources must be available to deliver on the contract.

Design verification

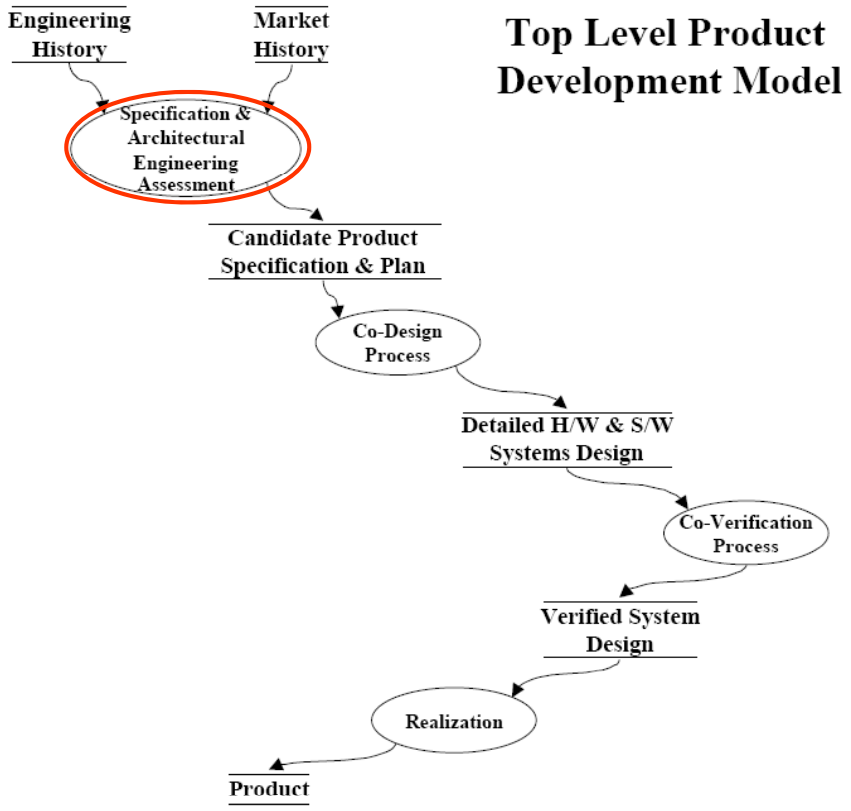
- Verifying multiprocessors is hard:
  - Observe and control data.
  - Drive part of the system into a desired state.
  - Generate and test timing effects.
- CoMET simulator
  - Hellestrand

Engineering Systems on a Chip:

The Bloody Revolution in Tools, Methodologies and Power

Graham R. Hellestrand

The dismantling of the power base in the semiconductor divisions of the major electronics companies to accommodate software engineering and systems integration on an equal footing with hardware engineering is a battle in full flight. Like the Iliad, where Greek's fiercely and intermittently battled Trojans for ten long years, the outcome of the systems engineering battle is certain. Unlike the complete destruction of the city of Troy, the resulting realignment will be a triumvirate between – hardware, software and mechanical designers. The sooner the battle is won the sooner the potential of the three powerful potentates will yield novel systems and architectures to dazzle the techno-enervated masses. Like all power-sharing structures, it is unstable but ultimately governable by the dour and pragmatic economics of survival. Already fleet-footed start-up companies are demonstrating the fecundity of the new godhead – carpe diem!



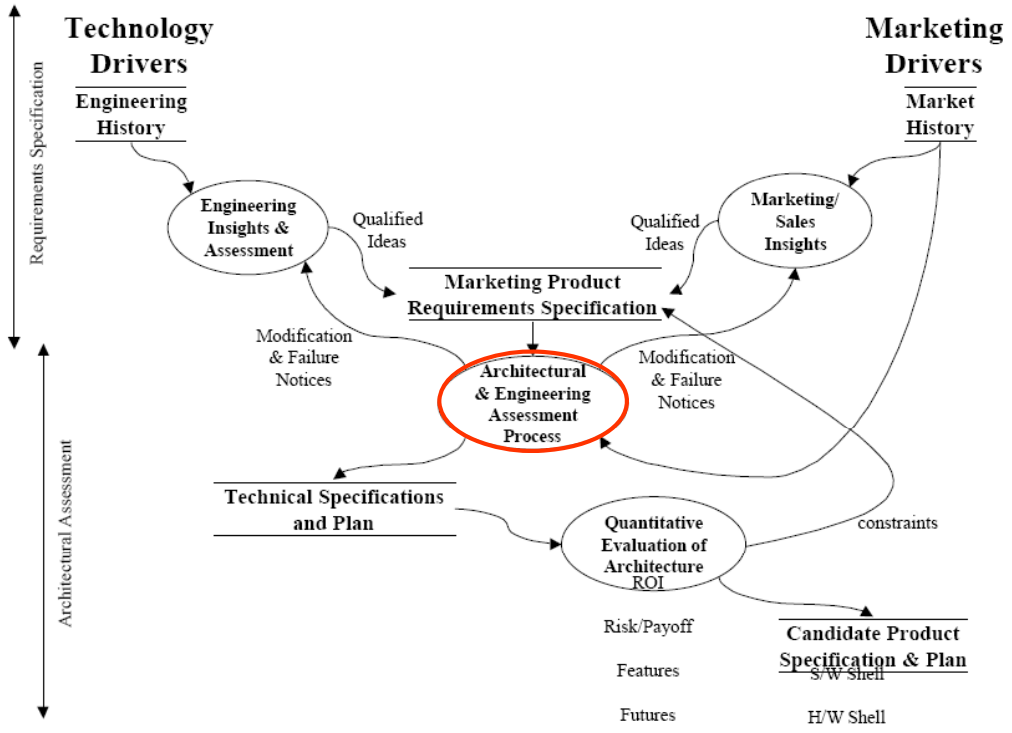


Figure 2: Specification and Architectural Engineering

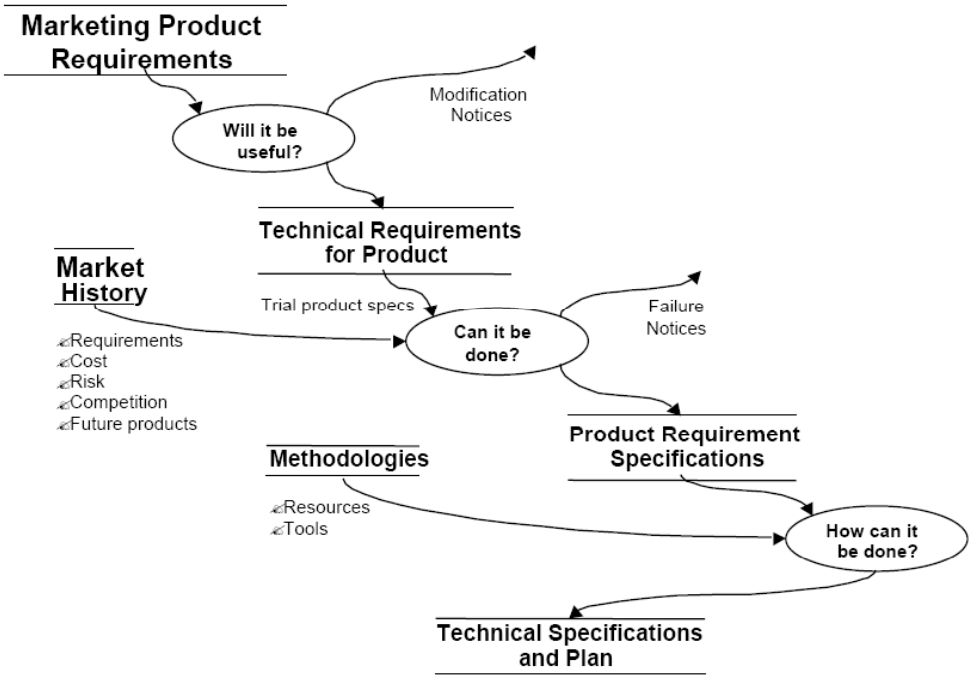


Figure 3: Architectural and Engineering Assessment

CoMET simulator

- A new, fast, and accurate processor model
- Two parts
  - Models for the behavior of instruction execution
  - Models for the dynamic parts of the processor
- Virtual processor model (VPM) describes function of the application running on the processor.
  - The code executed by a VPM may be C or C++ code
  - Very fast, like static timing analysis
- I/O parts that communicate between the processor and the hardware
  - Interrupts, cache, virtual memory, DMA, bus signals.
  - Speed is limited by the level of detail modeled and by the speed of the hardware simulator effecting communication
- Simulation backplane connects processor models and hardware models.
  - Many VPMs and a single HDL simulator
  - The back plane kernel mediates the maintenance of causality between the domains that defines their own relative frames of space and time

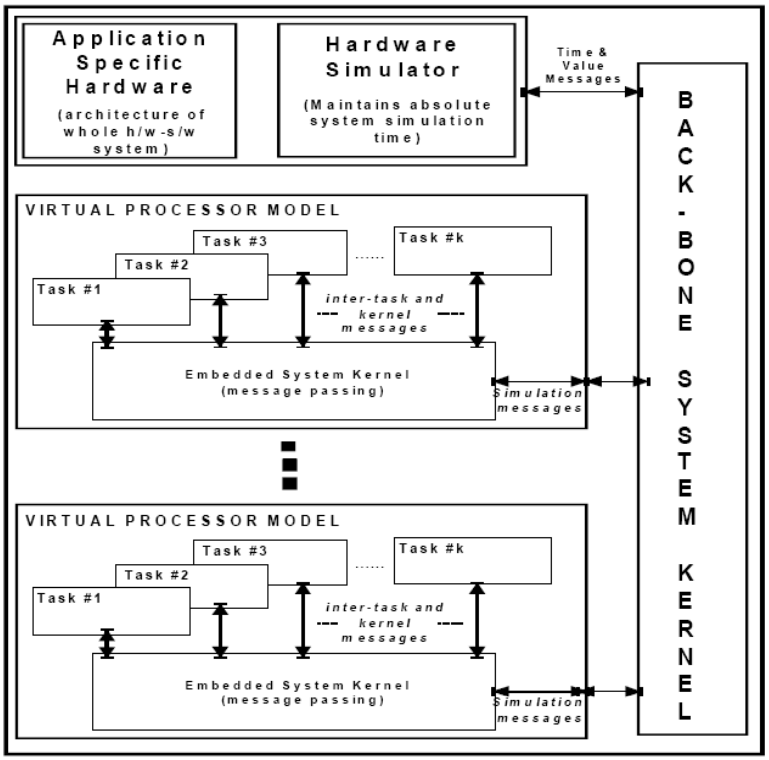


Figure 4. A Hardware-Software System Simulator Model

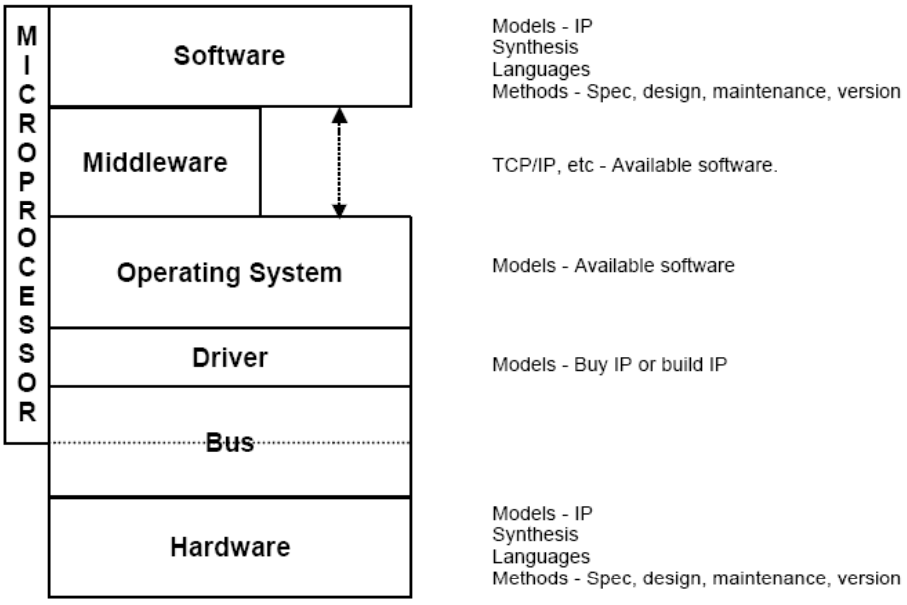


Figure 5. Components of a Typical System

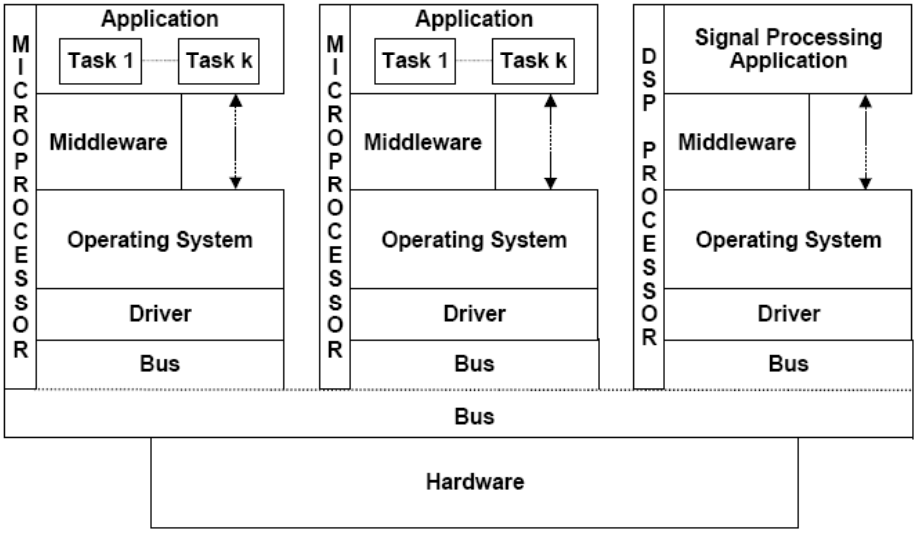


Figure 6. Multi-processor System Model