Appeared in IEEE Computer, February 1995

RPM: A RAPID PROTOTYPING ENGINE FOR MULTIPROCESSOR SYSTEMS[1]

Luiz Andre Barroso, Sasan Iman, Jaeheon Jeong,
Koray Öner, Krishnan Ramamurthy and Michel Dubois

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562
(213)740-4475
dubois@paris.usc.edu

Abstract

In multiprocessor systems, processing nodes contain a processor, some cache and a share of the system memory, and are connected through a scalable interconnect. The system memory partitions may be shared (shared-memory systems) or disjoint (message-passing systems). Within each class of systems many architectural variations are possible. Fair comparisons among systems are difficult because of the lack of a common hardware platform to implement the different architectures.

RPM (Rapid Prototyping engine for Multiprocessors) is a hardware emulator for the rapid prototyping of various multiprocessor architectures. In RPM, the hardware of the target machine is emulated by reprogrammable controllers implemented with Field-Programmable Gate Arrays (FPGAs). The processors, memories and interconnect are off-the-shelf and their relative speeds can be modified to emulate various component technologies. Every emulation is an actual incarnation of the target machine and therefore software written for the target machine can be easily ported on it with little modification and without instrumentation of the code.

In this paper, we describe the architecture of RPM, its performance and the prototyping methodology. We also compare our approach with simulation and breadboard prototyping.

Keywords: Field-Programmable Gate Arrays (FPGAs), message-passing multicomputers, shared-memory multiprocessors, design verification, performance evaluation, simulation.

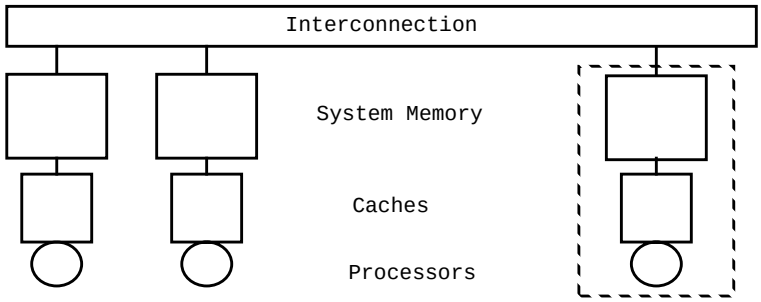
---

1. This work is supported by the National Science Foundation under Grant MIP-9223812
Future developments on this project will be posted on WWW at http://www.usc.edu/dept/ceng/dubois/RPM.html

1. INTRODUCTION

Multiprocessor systems are becoming common place in the computing industry. The consensus among machine designers is to favor asynchronous MIMD (Multiple Instructions Multiple Data streams) systems, in which processors execute their own instructions and run on different clocks. In these systems, processing elements contain a processor, some cache and a share of the system memory, and are connected through a scalable interconnect which facilitates machine packaging, such as a bus or a mesh (see Fig. 1). Whereas this physical model dominates, disagreement exists as to the interprocessor communication mechanism. There are two dominant models. One is based on disjoint memories and message-passing and the other is based on shared-memory. In a message-passing system, processors communicate by exchanging explicit messages through *send* and *receive* primitives. A received message is put into a local buffer, which is accessed by the processor when it executes a receive. In the shared-memory model, processors communicate through load and store instructions and some form of explicit synchronization among processors is required at times to avoid data access races [6].

FIGURE 1. Physical organization of most MIMD systems



The shared-memory model facilitates fine grain (word level) communication but requires a large number of instructions to transmit large chunks of data, whereas the message-passing model can transmit large amounts of data in a single message. In terms of ease of programming, the shared-memory model has been so far the favored transition path from uniprocessors to multi-processors. On the other hand, message-passing systems are generally perceived as more scalable than shared-memory systems. One concern in both kinds of systems is the growing disparity

between the processor speed and the speed of communication. In message-passing systems, messages have traditionally suffered from a large software overhead, which explains the comparatively large latencies of sends and receives as compared to the latencies of loads and stores in shared-memory machines. Ways to combat these large latencies are to implement light-weight, hardware-based message primitives and to overlap message passing with computation. In shared-memory systems, the large latencies of loads and stores on shared data is also a problem, which is usually solved by complex shared-memory access mechanisms.

Some researchers advocate private caches [6] [14] with hardware- or software-based consistency maintenance, such as in the Stanford DASH [11] prototype. The consistency protocol, the constraints on the ordering of memory accesses [6], the cache parameters, as well as the interconnect latency and bandwidth are all factors affecting the performance and programming ease of multiprocessors. Machines such as the DASH have been called CC-NUMA (for Cache-Coherent Non-Uniform Memory Access) architectures to differentiate them from COMAs (Cache-Only Memory Architectures) exemplified by the Data Diffusion Machine (or DDM) [8]. A COMA has the same architecture as the one shown in Fig. 1 and communication is done through shared variables; however, no main memory is present and, instead, the memory in each processor node implements a huge cache called *attraction memory*.

There is a trend today towards integrating the message-passing and shared-memory paradigms in order to draw from the strengths of both [10]. The implementation of message-passing on top of shared-memory is straightforward but special hardware is needed to make it efficient. Recently [12] it was shown that shared-memory can be implemented on top of a message-passing system. It is widely expected that future multiprocessors will be hybrid systems, supporting both the shared-memory and message-passing paradigms. To understand which form such an integrated system should take comparisons among systems are required. Presently, comparisons are difficult to make and hard to validate because of the lack of a common hardware platform to implement the different models.

Because multiprocessors are complex and powerful the correctness of a design and its expected performance are very difficult to evaluate before the machine is built. Traditionally two approaches have been taken to verify a design: breadboard prototyping and software simulation. A breadboard prototype is costly, takes years to build and explores a single or a few design points. Discrete-event simulation is very flexible but very slow if the design is simulated in details and it is subject to validity problems because the target system must be considerably abstracted in order to keep simulation times reasonable. In some industrial projects where a detailed and faithful simulation of a target system has been done in software, the simulation runs at the speed of a few cycles of the target system per second of simulation. The parallelization of discrete-event simulation is an ad-hoc procedure and usually exhibits low speedup [7]. Most simulators [5] [13] rely on the direct execution of each target instruction on the host and, because the code (either source or binary) must be instrumented, it is difficult to simulate efficiently the execution of interesting workloads, other than scientific programs with little or no I/O.

The major objective of the RPM project is to develop a common, configurable hardware platform to emulate faithfully the different models of MIMD systems with up to eight execution processors. Emulation is orders-of-magnitude faster than simulation and therefore an emulator can run problems with large data set sizes, more representative of the workloads for which the target machine is designed. An emulation is closer to the target implementation than an abstracted simulation and therefore more reliable performance evaluation and design verification can be done. Finally, an emulator is a real computer with its own I/O and the code running on the emulator is not instrumented. As a result, the emulator “looks” exactly like the target machine to the programmer and a variety of workloads can be ported on the emulator, including code from production compilers, operating systems, and software utilities.

In the following, we first introduce the hardware emulation approach for multiprocessors. Then we describe the architecture of RPM. In Sections 4 to 6 we explain the methodology to keep track of emulated time, to measure performance, and to program an emulation. Finally, in Sec-

tions 7 and 8 we show the expected performance of RPM and we compare the emulation approach to simulation and breadboard prototyping.

2. THE HARDWARE EMULATION APPROACH

At the University of Southern California we have been experimenting for a year with a new approach to the rapid prototyping of multiprocessor systems. The approach is based on *hardware emulation*. Emulators have been used in the past to experiment with instruction sets. At the time when most processors were microcoded, it was convenient to have a machine which could execute various instruction sets in order to run software developed for different machines. Moreover, emulators were used in the development of new instruction sets in order to quickly obtain an implementation, to verify the correctness of the instruction set and to start writing the software so that both the machine and the software could be ready at the same time. To the best of our knowledge, no attempt to apply the emulation approach to the design and verification of multiprocessor systems has ever been made.

Several technologies, including FPGAs (Field-Programmable Gate Arrays), and efficient Computer-Aided Design (CAD) tools, are currently converging, making it possible to build and program flexible multiprocessor emulators. Additionally, open software and hardware standards, and the existence of services for the rapid design and fabrication of printed-circuit boards facilitate the design of low cost multiprocessor emulators in a short time.

2.1. FPGAs (Field Programmable Gate Arrays)

Field-Programmable Gate Arrays (FPGAs) [15] are high-density, user configurable ASIC devices. FPGAs are in-circuit programmable by software. They have evolved over the past 15 years from simple chips that could replace a small number of gates (300 gates in a 20-pin package) to complex arrays which can implement large circuits of more than 20,000 gates in more than 200-pin packages. Meanwhile maximum operating speeds have soared from 20 to over 100 MHz. The current trend for FPGA technology is expected to continue unabated for some time to

come. To take advantage of this rapid technological improvement and to increase the lifetime of their design, designers must engineer the rest of their board for the highest possible clock rate and take advantage of the pin compatibility of new FPGA products.

These recent advances in FPGA technology have turned hardware emulation into a more practical design verification/analysis technique. Solutions for hardware emulation have been made available by commercial companies in the form of turnkey solutions. Turnkey solution vendors such as *Quickturn* [16] use generalized prototype boards where all programmable devices and interconnects are pre-placed in a fixed manner. Tightly integrated software takes care of logic mapping and partitioning of multiple FPGAs. Prototypes of new microprocessors such as the Intel Pentium have been built and debugged with their actual software, using the *Quickturn* Enterprise Emulation System. FPGAs have also inspired several research projects, such as the SPLASH attached processor, a reconfigurable systolic array built at the Supercomputing Research Center [1].

2.2. Efficient CAD Tools

The new generation of design automation tools allows designers to move to a higher level of abstraction. Designers no longer need to deal with the gate-level design of digital circuits. They can specify their design at the Register Transfer Level (RTL) using High-level Description Languages (HDL) such as VHDL and Verilog. The availability of such tools allows designers to complete more complex designs in a much shorter time frame. In particular, the programming of FPGAs is greatly simplified. Testing and simulation tools also help designers to verify the functionality of their design before it is implemented. Integrated tool sets that encapsulate all such design automation technologies provide a fast and reliable methodology to implement and verify a proposed design.

2.3. Open Standards

Open standards for instruction sets, software systems and interconnects make it possible to obtain

easily and inexpensively the key components making up a multiprocessor system. For example, the processors in RPM use the SPARC instruction set [3] for which we have the detailed functional simulator from Sun Microsystems. The chip set from Newbridge implements the Futurebus+ standard for the interconnection and relieves us from developing our own bus. In the future, when we port system code, we can use a standard for which we can acquire the source code.

2.4. Services for the Rapid Design and Fabrication of Custom PC-boards

We have used the services of EZFAB at the University of Southern California Information Sciences Institute (ISI). EZFAB is part of the ARPA-sponsored Systems Assembly Project. Our task was limited to providing a correct netlist. EZFAB designed the board layout and placement and produced all the specifications needed by the board manufacturer in the form of a gerber format output. As a result, our involvement in board design and fabrication was minimal.

2.5. Our Emulation Approach Using FPGAs

RPM is built mostly from off-the-shelf components (including processors, SRAMs, DRAMs, FIFOs, bus interface and drivers, and backplane), but the cache, memory, coherence and communication controllers are built with FPGAs. The emulation of a particular machine model is done through the FPGAs and a part of the memory to which they are attached. Fig. 2 shows a picture of the machine.

The clock rate is 10MHz, which is about 10 times slower than the rate permitted by current board and PLD technologies (estimated at 100MHz). This compromise on the emulation efficiency results from two trade-offs. First, the design and fabrication of the PC-boards of the emulator are greatly simplified, because the boards are at the mature end of the technology curve. Second, the lower clock rate facilitates the configuration of the FPGAs. FPGAs are slower than other programmable logic devices or custom circuits – especially when they are programmed with VHDL synthesizers, which are often less than optimum but which greatly reduce the design time – and clocking them at low speed promotes the mapping of more complex circuits. In order to

have better flexibility, to emulate complex mechanisms and to further simplify the design, each processor clock (pclock) is emulated in several clocks. (Currently one pclock is eight clocks, but this number can be changed.) So, overall, the emulator currently runs 80 times slower than the target system which could be built with the best current technology. This low processing rate allows us to use a standard interconnection fabric and still have enough bandwidth to emulate useful interconnections.

FIGURE 2. RPM is made of nine processor boards connected to a standard Futurebus+ backplane

3. ARCHITECTURE OF RPM

The architecture of RPM has been geared towards the evaluation of multiprocessors with the general architecture shown in Fig. 1. The interconnections of the possible targets are limited to FIFO (First-In-First-Out) interconnections with uniform access latencies such as crossbars or busses; however, other interconnections, such as rings, can be modeled approximately. A FIFO interconnection is an interconnection such that messages sent between two nodes arrive at the destination in the same order in which they are sent.

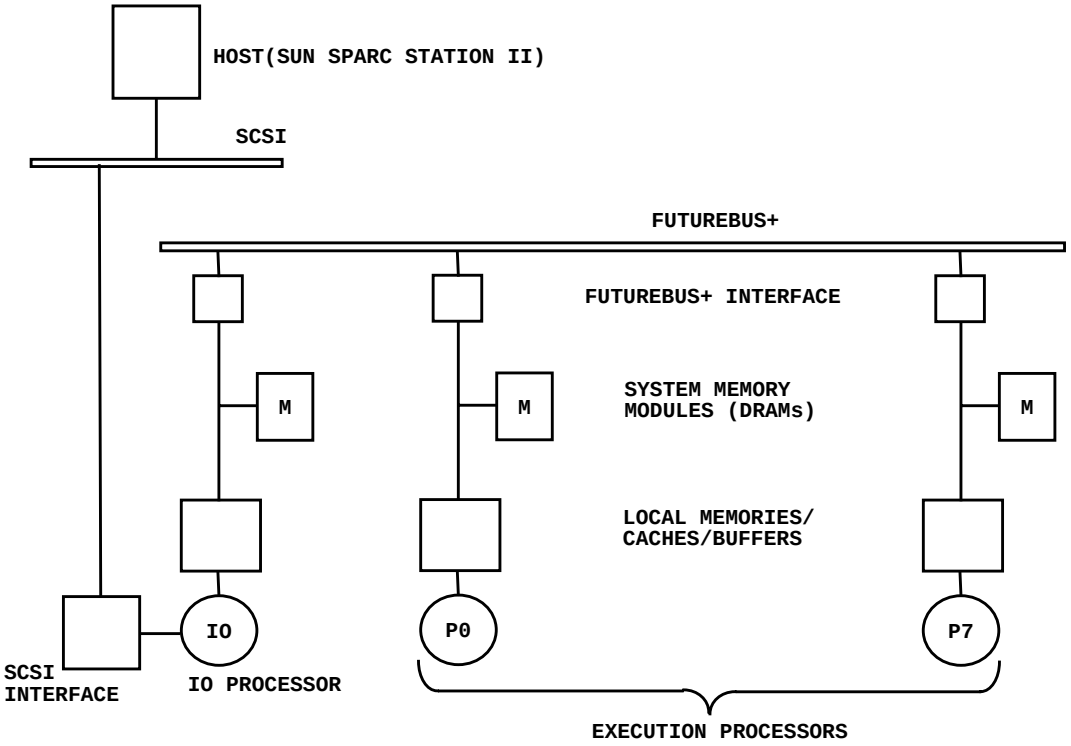
3.1. Hardware Organization

The hardware organization of RPM is shown in Fig. 3. It is made of nine identical boards, each with one SPARC processor (i.e., eight execution processor boards and one I/O processor board),

connected to a Futurebus+ backplane. The Futurebus+ is 64-bit wide (data) and is used in RPM as a medium to transmit packets among processor nodes. It also supports data broadcasting and interprocessor interrupts. The arbitration is distributed and takes between 200 and 600 nsec. The peak transfer rate is 20 Mwords/second or 80 Mbytes/second.

The number of clocks in each processor clock (pclock) is variable (it depends on the complexity of the mechanisms to emulate) but it is currently set to eight clocks, which yields a peak emulation rate of 10 MIPS (i.e., eight processors at 1.25 MIPS each) of the target system or 1.25 million cycles of the target per second. An I/O processor whose configuration is identical to the configuration of the execution processors is connected to a SUN SPARCStation 2 through a SCSI interface [3]. This workstation serves as the console for RPM and, additionally, executes its I/O requests. The peak I/O bandwidth is 1.25 Mbytes per second, which is more than sufficient for a 10 MIPS machine.

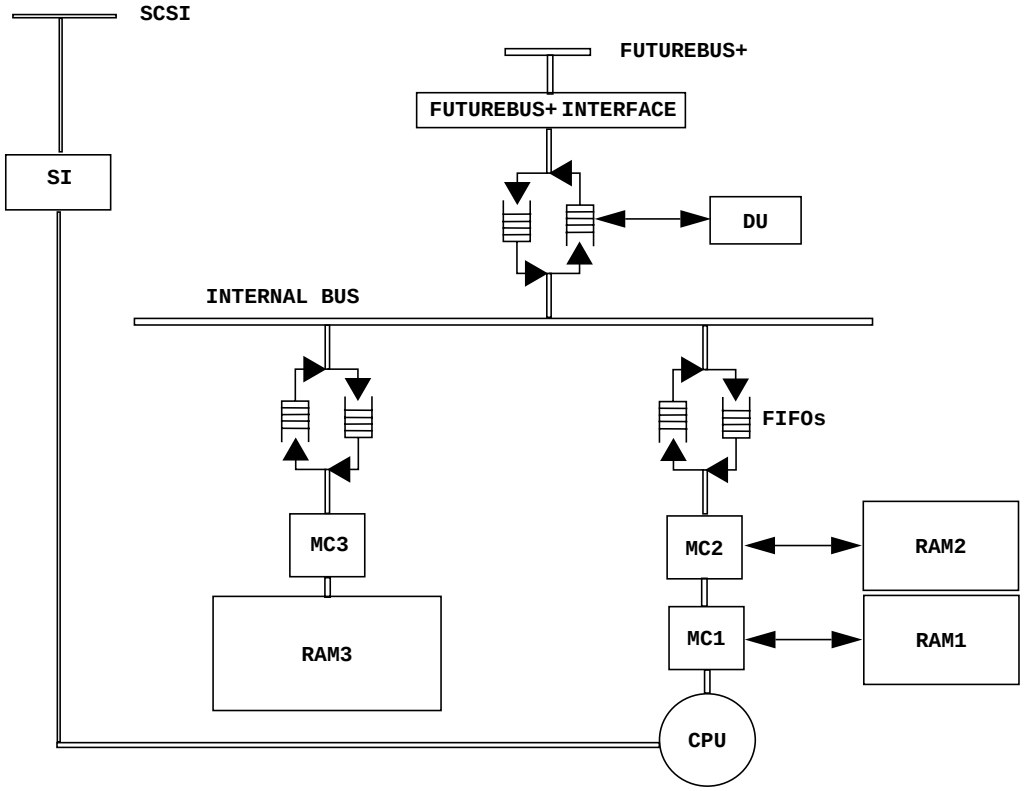
FIGURE 3. Overall configuration of RPM



The architecture of each processor board is shown in Fig. 4. The nine processors are LSI

Logic L64831 SPARC IU/FPU [3]. These single-chip processors can be clocked at up to 40 MHz and execute both integer and floating-point instructions. They have no on-chip cache and therefore all instruction fetches and data accesses are visible on the pins of the chip.

FIGURE 4. Block diagram of each processor node



There are three memories on each board, each controlled by a set of Xilinx FPGAs:

• MC1/RAM1: Each processor is attached to 2 Mbytes of static RAM (RAM1) controlled by two Xilinx XC4013[2] (called MC1). The major function of MC1 is to control the cycle-by-cycle execution of the processor. At the beginning of each processor access, MC1 blocks the processor, executes the sequence of steps needed to satisfy the access and unblocks the processor when the access is completed. MC1 also manages RAM1 as a cache and interacts with the second-level memory. MC1 implements extensions to the SPARC instruction set such as shared and exclusive

2. Each XC4013 contains the equivalent of 13K gates. We plan to upgrade to the Xilinx XC4025. This upgrade will double the number of equivalent gates in each controller. The XC4025 is pin-to-pin compatible with the XC4013.

prefetches. These extension are possible through SPARC's ASI (Alternate Space Identifiers) or through unused address bits. MC1 can also remap addresses in many different ways, including a full virtual-to-physical translation through a TLB.

• MC2/RAM2: An additional 8 Mbytes of static RAM (RAM2) controlled by three Xilinx XC4013 called MC2 makes up the second-level memory. MC2 interfaces the processor to the rest of the system and usually acts as a second-level cache controller.

• MC3/RAM3: System memory is emulated by 96 Mbytes of dynamic RAM (RAM3) controlled by two Xilinx XC4013 and one Cypress CYM7232 DRAM controller (called MC3).

The internal bus is a synchronous bus with a protocol similar to the Sun Microsystems MBUS protocol [3]. It is a 32-bit wide packet-switched bus which transfers packets of size between 16 and 128 bytes. Controllers MC2 and MC3 connect to the internal bus through very large, two-way FIFO buffers, which are there to prevent deadlocks and to relieve the controllers from managing the data transfers. All on-board datapaths are 32-bit wide.

The Delay Unit (DU) is a programmable unit which emulates variable interconnection delays. It is built with a FIFO controlled by one AMD MACH 210 chip. The FIFO (8 kbytes) contains blocks and messages which are sent to the bus interface after a programmable delay depending on the target machine's interconnect latencies and packet size. This delay is computed by the formula:

$$\text{Latency} = T_{\text{start}} + (\#\text{ of words in packet}) \times T_{\text{word}}$$

The Futurebus+ interface is made of off-the-shelf chip sets. It includes bus transceivers, plus the LIFE chip from Newbridge and a distributed arbiter chip from National Semiconductors. The processor speed is low relative to the bandwidth available on the Futurebus+. We have run simulations of RPM configured as a CC-NUMA architecture using the SPLASH benchmarks[3] with extremely small data set sizes (and therefore a high communication-to-computation ratio).

3. The SPLASH benchmarks are scientific benchmarks commonly used to compare the effectiveness of architectural features of multiprocessors (see for example the study in [4]).

With eight clocks per pclock, the bus utilization is lower than 10% even under the worst-case conditions. Interconnection traffic between any pair of processors can be measured in order to observe whether a particular switch in the target system's interconnect could be a hot spot. Note that the contention for the caches, the internal bus and the memory inside each processor node is modeled in full detail. A picture of a board is shown in Fig. 5.

FIGURE 5. Each board of RPM is a 22"x18" 10 layer PC board

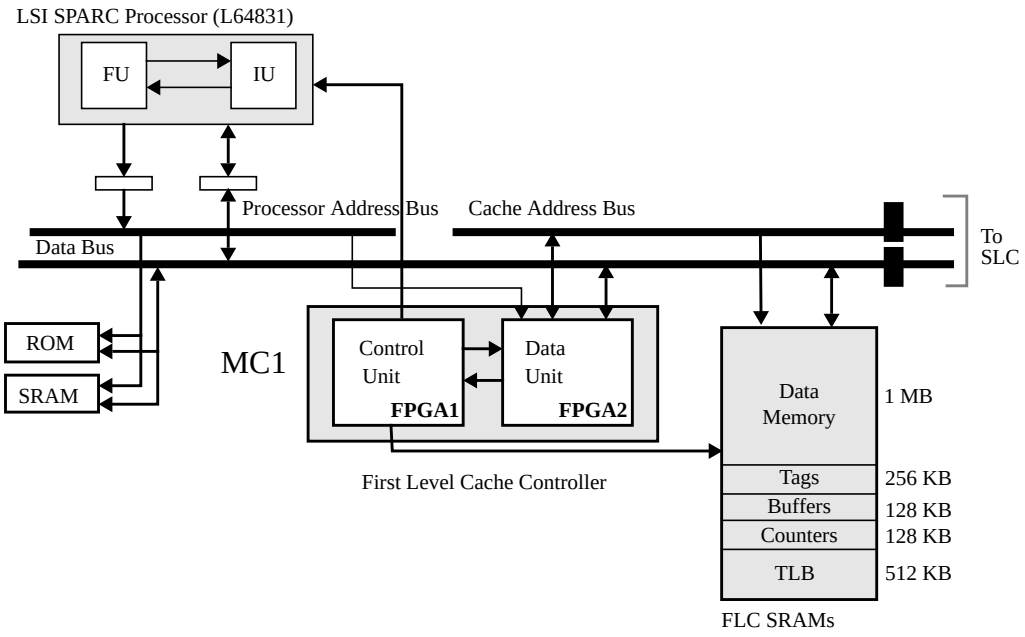
3.2. Emulation of CC-NUMAs with Central Directory Protocol

The first emulator that we have developed is a system with hardware-enforced cache-coherence under strong or weak ordering of memory accesses [6]. The protocol is directory-based and each memory block has a home node where a directory records the presence and state of copies in every cache [11] [14]. The memory and the cache directories have pending states so that transactions for different blocks are executed concurrently. In this emulator, MC1/RAM1 is a first-level write-through cache (containing both data and instructions), MC2/RAM2 is a second-level write-back cache and MC3/RAM3 is the main memory.

Non-blocking prefetches are supported in the second-level cache. These non-blocking accesses are issued by the compiler to direct the second-level cache to prefetch cache blocks before the data is needed. A virtually unlimited number of prefetches can be pending at any time in the second-level cache. When a store is issued by the processor the store is always propagated

to the second-level cache. If the store misses in the first-level cache no block is allocated (no allocation on store misses). Under strong ordering of memory accesses [6] (enforcing sequential consistency), the first-level cache and the processor block on each write access which misses or which requires coherence activity in the second-level cache. Under weak ordering of memory accesses there can be a write buffer between the first- and second-level caches (first-level write buffer) and between the second-level cache and the internal bus (second-level write buffer). The second-level write buffer can be assisted by a write cache (WC), which is a small cache keeping track of partially modified blocks [4]. The first protocol is a pure write invalidate protocol but we intend to implement write-update and competitive-update protocols as well as hardware-based prefetching as described in [4].

FIGURE 6. The processor and its first-level cache

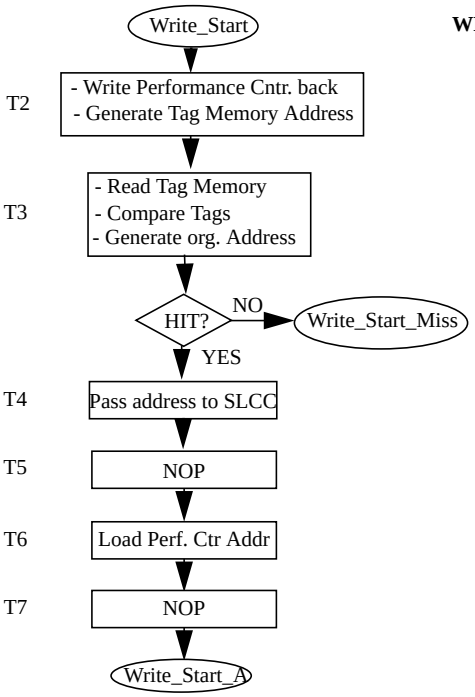


The emulation of each pclock is implemented by a combination of control in the FPGAs and buffer space in the RAMs attached to them. This buffer space supports the emulation of special memories, such as cache directories, a TLB (Translation Lookaside Buffer), which is part of the hardware for virtual-memory support [3], or write buffers between the first- and the second-level caches and between the second-level cache and the internal bus. Additional memory space

(called *count memory*) is reserved for event counters (see Section 5).

The SRAM implementing the **first-level cache** (FLC) is divided into five parts (see Fig. 6): the data memory (up to 1 Mbyte), the cache directory, the Translation Lookaside Buffer (TLB), the space for the emulation of prefetch and write buffers, and the space dedicated to the collection of performance statistics. The controller is partitioned across two FPGAs: one for the control unit and the other for the data unit. Currently the first-level cache is write-through and direct-mapped with a block size of 16 bytes.

FIGURE 7. Partial flowchart and its VHDL code for a write cycle in the first-level cache



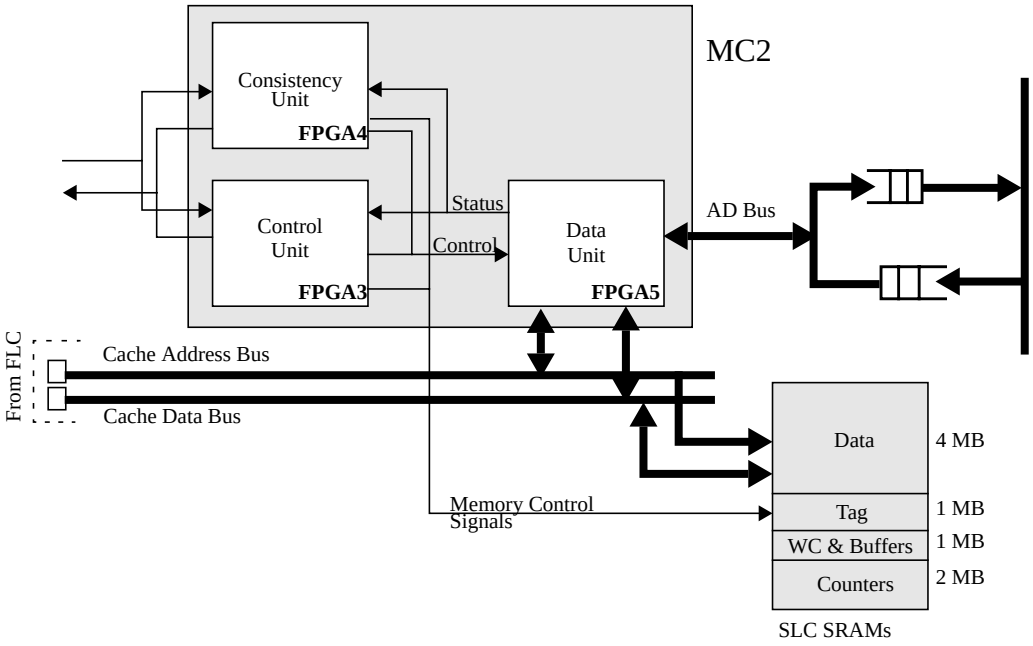

```
WHEN wrt_start_st =>
   CASE (Time) IS
      WHEN T2 =>
        prep_tag_org(n_select_addr, n_flc_section, n_load_a_c_reg);
        prep_r_w(WRITE,WORD,n_flc_mem_access, n_read_out, n_a_c_oe, n_size_out);
      WHEN T3 =>
        prep_r_w(READ,WORD, n_flc_mem_access, n_read_out, n_a_c_oe, n_size_out);
        IF (is_hit_org = HI) THEN
            n_hit <= HI;                          -- Normal write HIT!!
        ELSE
            n_state <= wrt_start_miss_v1d;        -- Normal write MISS!!
            n_miss_req <= HI;
            n_hit <= LOW;
        END IF;
        prep_org_addr(n_select_addr, n_flc_section, n_load_a_c_reg);
      WHEN T4 =>
        n_a_c_oe <= LOW;
        n_a_cpfs_en <= HI;                        -- Pass Address to SLCC
      WHEN T5 =>
        NULL;
      WHEN T6 =>
        load_perf_addr <= HI;
      WHEN T7 =>
        n_state <= wrt_start_a_v1d;
      WHEN others =>
        NULL;
   END CASE;
```

A typical data read cycle in the first-level cache consists of receiving an address from the processor, translating the address in the TLB space, accessing the cache directory space, fetching the data from the cache data space, fetching the counter in count memory for that event, updating the counter and returning the data to the processor (all this is done in eight cycles). Fig. 7 shows a partial flowchart for a write access in the first-level cache and its VHDL implementation. (This sequence has no TLB access.) This sequence emulates one pclock in the processor write cycle. As

can be seen, the updates of event counters in count memory are implemented within the control sequence and are pipelined. T0 and T1 are missing in the flowchart because they are common to all accesses.

The **second-level cache** (SLC) is implemented by MC2/RAM2. The current configuration is a two-way set-associative write-back cache and a block size of 16 bytes (see Fig. 8). Half of the memory (up to 4 Mbytes) is for the second-level cache data memory. The other half is dedicated to the cache directory, various buffers (second-level write buffer and write cache) and count memory. The second-level cache controller is by far the most complex controller of the machine. It is implemented by three FPGAs: the data unit contains the hardware resources needed by the controller, including the buses to the RAM; the control unit implements the basic cache control functions and the consistency unit contains additional control to enforce memory access ordering and to manage the write buffers and prefetching hardware [4].

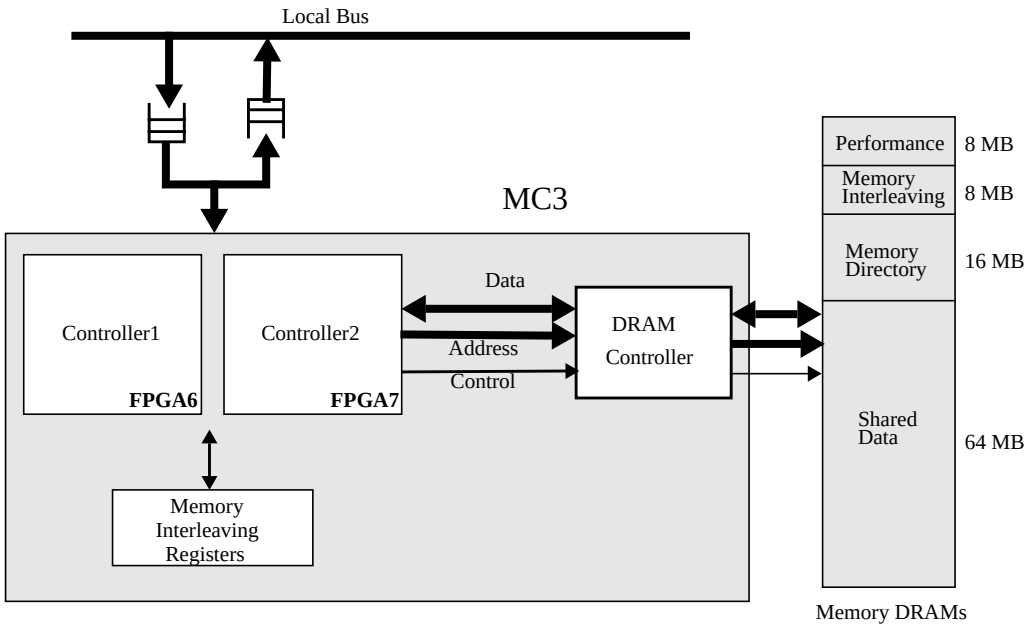
FIGURE 8. The Second-Level Cache Architecture.



The **main memory** is implemented in MC3/RAM3. 64 Mbytes are devoted to the data memory, 16 Mbytes to the directory (one 32-bit word per 16-byte block), 8 Mbytes for perfor-

mance counters and 8 Mbytes to support the emulation of various hardware mechanisms, including the *virtual interleaving* of on-board memory (Fig. 9). If current trends continue and the speed gap between processor and DRAM keeps widening, the conflicts at the memory modules will be so high that some form of on-board memory interleaving will be required. (By “interleaving” we mean that several independent memory banks with their own controllers can process memory requests concurrently.) In RPM, this interleaving effect is obtained by multiplexing in time the memory controller. *Virtual interleaving* of memory relies on the fact that a large number of cycles are available in the emulator to emulate memory transactions and is supported by a set of eight interleaving registers (for up to eight interleaved banks) and some buffer space in memory. Each interleaving register contains a counter which is decremented at every pclock. When the counter reaches zero the memory bank is free. A request to a busy bank is queued at the controller.

FIGURE 9. The main memory and its controllers



A typical memory transaction in RPM has three phases: *prelude*, *suspension* and *completion*. During the prelude the packet is received and decoded and the directory is accessed to find the state of the block. To suspend the request, a transaction completion record containing all the information needed to resume and complete the request is stored in a memory location corre-

sponding to the memory bank and the interleaving register is filled with a value in pclocks corresponding to the suspension time. While the request is suspended, a different (free) virtual memory bank can be accessed. When the interleaving register reaches zero, the memory controller is interrupted; it then fetches the transaction completion record in memory and completes the transaction by (possibly) sending some messages (completion phase). Virtual memory interleaving is further illustrated in Section 4.

Many parameters can be changed easily in the system above, within limits. All cache and TLB parameters can be changed. Latencies and bandwidths of various components can be changed. We can explore latency tolerance techniques such as non-blocking second level cache with write buffers and write caches as well as data prefetching hardware. We can experiment with various directory structures [14]. We can change the protocols (e.g., write-invalidate, write-update, or competitive-update), or we can implement multiple protocols and apply different protocols to different blocks.

Table 1: FPGA statistics

Controller	FPGA Name	Occupied CLBs (%)	Packed CLBs (%)	# of IO pins used
MC1	Control Unit	73	47	122
	Data Unit	60	41	176
MC2	Control Unit	82	54	118
	Consistency Unit	not used	not used	not used
	Data Unit	95	73	171
MC3	Contro1ler1	89	65	141
	Controller2	66	47	95

3.3. FPGA Statistics

Table 1 gives the current statistics on the utilization of the FPGAs. (These statistics are for strong ordering of memory accesses with no latency tolerance hardware except for prefetch support in the second-level cache.) CLBs or Configurable Logic Blocks [15] are the logic building blocks in

the Xilinx XC4013 FPGAs. Each block contains hardware to implement random logic. A CLB is packed when all of its logic is used and it is simply occupied when any of its logic is used. Currently, the consistency unit is not programmed. Finally, the total number of programmable I/O pins in the XC4013 is 192. As can be seen, the CLB utilization of a few FPGAs are dangerously close to their maximum.

The current designs are far from optimum. We derived them as fast as we could to obtain a first working system. We expect that more careful designs will reduce the CLB utilization for some of these FPGAs. We will also upgrade to Xilinx XC4025's as soon as they become available. This will double the capacity of each FPGA. Higher capacity implies that more complex or faster designs can be mapped. Finally, we can upgrade our synthesis tools to generate better circuit designs.

3.4. Emulation of Other Architectures

Given its generic board architecture, RPM is capable of emulating in full detail very complex target multiprocessors with very different architectures, provided they fit the generic block diagram of Fig.1. Examples are:

• **CC-NUMAs with Linked-List Directories**: Instead of a centralized directory located at the home memory, the directory is distributed by linking caches containing a copy of the block through a set of hardware pointers in each cache entry. This organization is adopted in the Scalable Coherent Interface (SCI) standard [9].

• **COMAs** (Cache-Only Memory Architectures): This is the architecture of machines such as the Data Diffusion Machine (DDM) [8]. In this case there is no system memory and MC3/RAM3 acts as a huge cache (also called *attraction memory*) and the DRAM directory is replaced by the *attraction memory state*. In this configuration, MC1/RAM1 and/or MC2/RAM2 may be configured as caches.

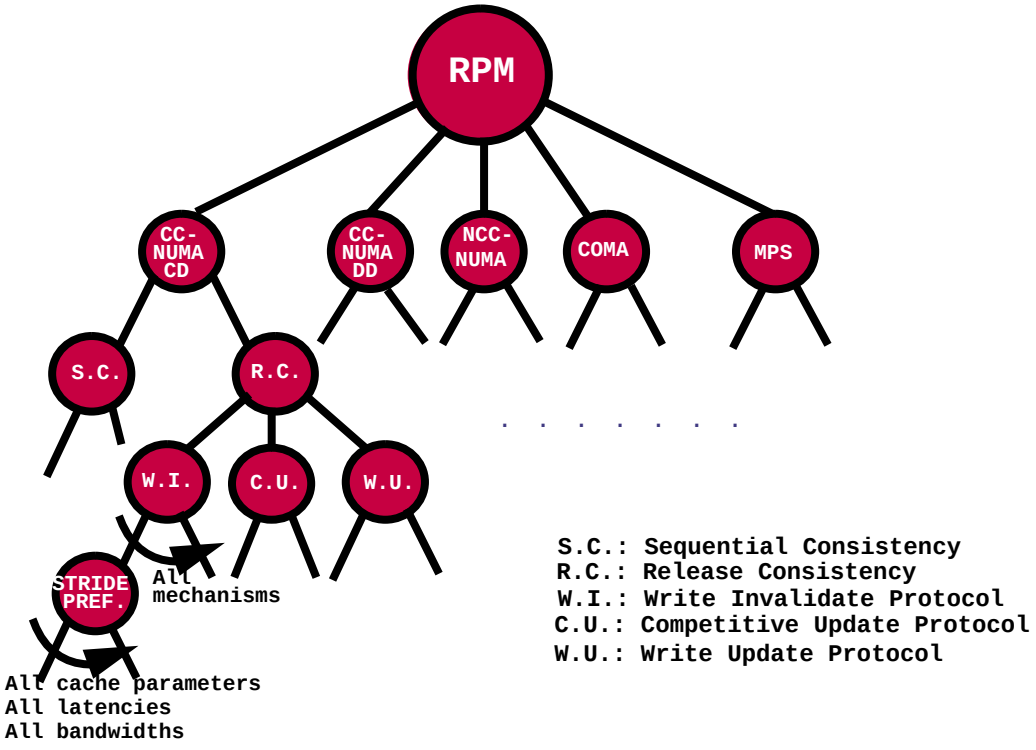
• **MPS** (Message-Passing Systems): In this configuration MC3/RAM3 acts as the local (private)

memory of the processor, MC2/RAM2 acts as a Message Passing Controller (MPC), and MC1/ RAM1 acts as the processor cache. The functions of the MPC are to buffer messages sent or received by the processor, to format out-going packets according to the protocol in the target, to decode the messages received, and to interrupt the processor when messages are received. This message-passing architecture can also include hardware primitives to support virtual shared memory efficiently [12].

• **Mixed Shared-memory and Message Passing Systems**: Every shared-memory organization can be augmented with a message-passing facility for bulk transfers of data among processors, as was done in Alewife [10].

Fig. 10 shows the genealogy of some of the possible emulators which can be derived from RPM (NCC-NUMA stands for “non cache-coherent NUMA”; the CC and CD appended to CC-NUMA stand for “centralized directory” and “distributed directory”).

FIGURE 10. The genealogy of emulators



4. KEEPING TRACK OF TIME: TIME SCALING

Since the speeds of the hardware emulation and of the target system are different, timings measured on the emulator must be related to the timings in the target machine. Rather than keeping track of simulated time through event-driven mechanisms and timestamping (as is done in software simulators [7]), time is *scaled*. *Time scaling* preserves the *relative* timing of components in the emulator and in the target, and absolute times in the target are derived from executions on RPM by simple scaling arguments. For example, it is intuitively obvious that the processor utilization in a system with processors running at 100 MHz and with average memory latencies of 100 nanoseconds is equal to the processor utilization in a system with the same architecture but with processors running at 1 MHz and with average memory latencies of 10 microseconds. All performance metrics can be scaled this way, and therefore we do not have to build the system with the most up-to-date and fastest technology provided we scale memory, interconnection and processor speeds appropriately.

Every component (interconnect, cache, memory and I/O processor) is characterized by two fundamental performance measures: latency and bandwidth. These two measures can be independent. For example, two networks can have the same latency but one may have more bandwidth because it has more links; similarly, the bandwidth of a memory can be increased (while its latency remains the same) by interleaving it. Another important factor is the width of the data paths. In RPM, all data paths on board are 32-bit wide and the system bus (Futurebus+) is 64-bit wide. We can emulate one cycle of a data path with 64, 128 or even 256 bits by 2, 4, or 8 cycles of the 32-bit data path in RPM.

A convenient unit for all timings is the *pclock* – the clock period of the processor. **If the latencies of all components are expressed in terms of pclocks and if all component bandwidths are expressed in terms of bytes per pclock, then systems with components of equal latencies and bandwidths are equivalent.**

In RPM, a pclock is currently eight cycles. This gives the emulator eight cycles to simu-

late all the activities occurring in one pclock in the target system. To simulate variable latencies, we delay requests. To simulate variable bandwidth of a given resource, we must vary the number of cycles that each request keeps the resource busy. For example, the latency of local memory can be increased by delaying the requests; its bandwidth can be decreased by inserting dummy cycles in the control sequence of the memory controller or it can be increased through memory interleaving.

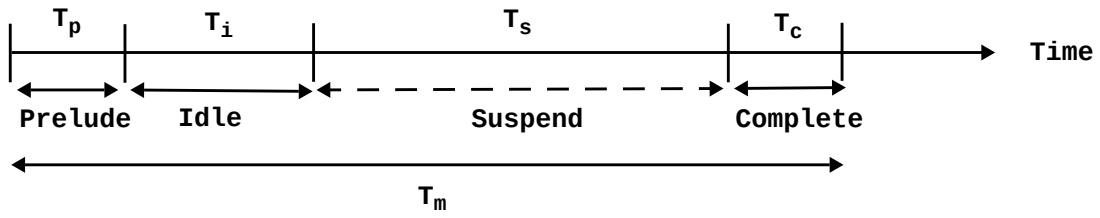
The latencies and bandwidths of the first-level cache and of the internal bus are not adjusted. The implication is that their speed scales up proportionally with the speed of processors in all target systems.

The Delay Unit (DU) simulates variable latencies in the interconnection (emulated by the Futurebus+). As configured, the bandwidth available on the Futurebus+ is very large. To run experiments under limited interconnect bandwidth, we can reduce the width of the bus (by reprogramming the LIFE chip) as well as artificially increase the size of each packet. To increase the interconnect bandwidth the number of clocks per pclock in each processor can be doubled or even quadrupled. The accuracy of measurements requires that the Futurebus+ traffic is monitored so that excessive traffic leading to significant conflicts and delays at the bus are detected.

To illustrate time scaling and virtual memory interleaving, consider the simple case where a miss occurs in the second-level cache, the block address maps to the local on-board memory (local miss), and the block is uncached elsewhere. In the target system, T_m is the latency of the miss and the memory is *n*-way interleaved so that the memory can deliver *n* blocks every T_m pclock. RPM has a monolithic memory controller, which must emulate the operations of the *n* parallel memory controllers of the target system. Therefore the memory transaction must keep the controller busy for T_m/n pclocks by idling the controller before suspending the miss request (see Section 3.3). On the other hand, the total latency of the miss in RPM must be T_m pclocks, the same as in the target system. Let T_p, T_i, T_s *and* T_c be respectively the prelude time, the idle time, the suspension time, and the completion time of the miss request in RPM. (All times are in

pclocks.) The various phases of a memory access in RPM are illustrated in Fig. 11. The memory controller is busy during the whole access except during T_s, when an access to the other memory bank can be accepted by the controller.

FIGURE 11. Phases of a memory access in RPM (for the example of Fig. 12)



T_p and T_c are known from the implementation of the emulation. To find T_i and T_s we have the following constraints:

$$T_m = T_p + T_i + T_s + T_c \quad \text{and} \tag{1}$$

$$T_m/n = T_p + T_i + T_c \tag{2}$$

(1) enforces the same latency in RPM and in the target whereas (2) enforces the same utilization of the controllers in both systems. Unknown T_i and T_s are then given by:

$$T_i = T_m/n - T_p - T_c \quad \text{and} \quad T_s = T_m\,(n - 1)/n \tag{3}$$

Consider a target system with the timing characteristics shown in Fig. 12 (left column). The corresponding configuration of RPM is specified in the right column. In this example, the memory of the target system is assumed to work in page mode: the access time to a 64-bit word takes 100nsec and the access time to a 128-bit block takes 140nsec in the target. Because the target pclock is 5 nsec and the pclock in RPM is 800 nsec, a 140nsec block access time in the target translates into 140/5 pclocks or 28x8 = 224 clocks in RPM. This is a huge number of cycles and therefore the memory controller can emulate very complex directory mechanisms during this time. The timing is tighter in the second level cache because it is usually built with SRAMs in the target system. In the example, the second-level cache has 4 pclocks or 32 clocks to emulate a

block transfer between second- and first-level caches. As the block size increases, the number of extra cycles available for emulation also increases.

FIGURE 12. Simple example illustrating time scaling

Target System:

- 200 MHz PROCESSORS; 1 PCLOCK = 5 NSEC
- INSTRUCTION FETCH: 1 PCLOCK
- LOAD: 2 PCLOCKS
- 100 NSEC (20 PCLOCKS) PER WORD ACCESS IN MEMORY
- 64-BITS DATA PATHS
- 16-BYTE BLOCKS
- MEMORY AND 2-ND LEVEL CACHE ARE 64-BIT WIDE
- 4 PCLOCKS PER MISS FROM SLC TO FLC
- 28 PCLOCKS PER BLOCK FETCH IN MEMORY
- MEMORY IS TWO-WAY INTERLEAVED
- MAX MEMORY BANDWIDTH: 1 BLOCK PER 14 PCLOCK

RPM Configuration:

- FLC INSTRUCTION FETCH: 8 CLOCKS
- FLC DATA FETCH: 8 CLOCKS
- 28 PCLOCKS OR 224 CLOCKS IN MEMORY PER MISS
- 4 PCLOCKS OR 32 CLOCKS AVAILABLE TO SLC TO TRANSFER BLOCK TO FLC.
- PRELUDE TIME: 30 CLOCKS
- COMPLETION TIME: 30 CLOCKS
- IDLE TIME: 112 - 30 - 30 = 52 CLOCKS
- SUSPENSION TIME: 224/2 = 112 CLOCKS

The scaling of I/O is very natural in RPM. The I/O bandwidth is 1.25 Mbytes per second. This bandwidth matches the I/O requirements of the eight execution processors, which have an aggregate peak processing rate of 10 MIPS. Note that when we simulate target systems with faster processors, we do not have to adjust I/O bandwidth because RPM always runs at the same speed, i.e. 10 MIPS peak whatever the target system speed is (provided one pclock= eight clocks). So, I/O bandwidth scales automatically. The other issue is I/O latency, which must be scaled. The service of I/O requests must be delayed as faster target processors are emulated. This scaling of I/O latency is done in software, with the support of an interrupt timer.

5. COLLECTING PERFORMANCE DATA

The primary mechanism to collect performance data involves event counters stored in a special area of each memory called *count memory*. In each of the three on-board memories, a set of counters keeping track of the occurrence of mutually exclusive events are updated every time a transaction is completed in the controller. Addresses of event counters are formed automatically by merging signals corresponding to basic events. These events are mutually exclusive, so that only one counter is updated at a time in each memory. Thousands of counters are present in each

memory, meaning that thousands of different events can be counted. At the end of the emulation run, the counters are uploaded and post-processed (basically they are selectively added together) to obtain the required performance data. This counting mechanism can be started and stopped under software control.

FIGURE 13. Generation of addresses for event counting in the first-level cache's count memory

Counter Address	Private/ $\overline{\text{Shared}}$	Read/ $\overline{\text{Write}}$	Hit/ $\overline{\text{Miss}}$	Basic Event
0	0	0	0	Shared-Write-Miss
1	0	0	1	Shared-Write-Hit
2	0	1	0	Shared-Read-Miss
3	0	1	1	Shared-Read-Hit
4	1	0	0	Private-Write-Miss
5	1	0	1	Private-Write-Hit
6	1	1	0	Private-Read-Miss
7	1	1	1	Private-Read-Hit

Fig. 13 illustrates how the addresses in count memory are generated for mutually exclusive events in the first-level cache. In this simple example, three signals – programmed in MC1– correspond each to one property of an access in the first-level cache. One signal indicates whether the access is to private or to shared data. The second line is the read/write signal from the processor and the third signal is high when the access hits and low when the access misses in the first-level cache. The combination of these signals forms a three-bit address, which can be used to address a counter in the area of RAM1 allocated to count memory. In this example there are only eight addresses, but in a practical situation up to 20 signals can be defined in each controller. For instance one signal may also distinguish between instructions and data, and, in the case of instructions, the opcode could also be a field of the address of the performance counters. (In this case, instruction types are histogrammed and, at the end of the program execution, we can obtain a dynamic instruction mix of the program by summing together the counters with addresses in the count memory having the same opcode field.) Several other special purpose monitoring strategies can be implemented in all levels of the memory hierarchy. They can be added to the basic count

memory mechanism or they can replace it, depending on the amount of hardware required.

6. PROGRAMMING RPM

In each board, seven FPGAs implement the controllers for the caches and the main memory. The behavior of each emulation is dictated by the protocols that these controllers execute. Therefore RPM is programmed by mapping the controllers for the target design into the FPGAs.

Currently, we do not have tools to partition automatically a design across multiple FPGAs. The data path and the RTL description of each controller are specified separately in VHDL. The netlist for each controller is generated in two steps. First an intermediate file in bliff format is produced from the VHDL description by Viewlogic's Viewsynthesis 2.2.1 tool and our interface tool. Second, the Berkeley SIS tool synthesizes this intermediate file into a netlist, which is then mapped into the FPGA using the Xilinx mapping tools.

Design parameters such as block sizes, cache sizes, latencies and bandwidth are specified as constants in the VHDL descriptions. Therefore the target machine parameters can easily be changed by modifying these constants in the designs and then recompiling the VHDL codes. This approach requires that each design is recompiled for each parametric change. An alternative approach would be to design the controllers such that the machine parameters are stored in registers inside the FPGAs. Changing parameters would only require that the content of these registers be changed. This approach would lead to slower and more complex designs, but would save time in recompiling the design for parametric changes.

In order to change the functionality of the target machine, controllers that implement different protocols have to be mapped into the FPGAs. Using the current approach, modifications to the functionality of each FPGA require the design and specification of the RTL description of each controller in VHDL. In reality, however, a large number of common functions are shared by all the possible designs for a given FPGA. Nevertheless, the task of re-programming the FPGAs is error-prone and increases the turnaround time for emulating different machines. In the future, we

expect that the turnaround time for this design stage will be reduced as behavioral compilers become more widely available. A behavioral compiler is a high-level synthesis tool which accepts an algorithmic description of a circuit to create the hardware. This algorithmic description could be derived directly from the high-level description of each protocol. In the short term we will develop libraries of parameterized designs for every possible configuration of each FPGA.

FIGURE 14. The three dimensions of flexibility in RPM

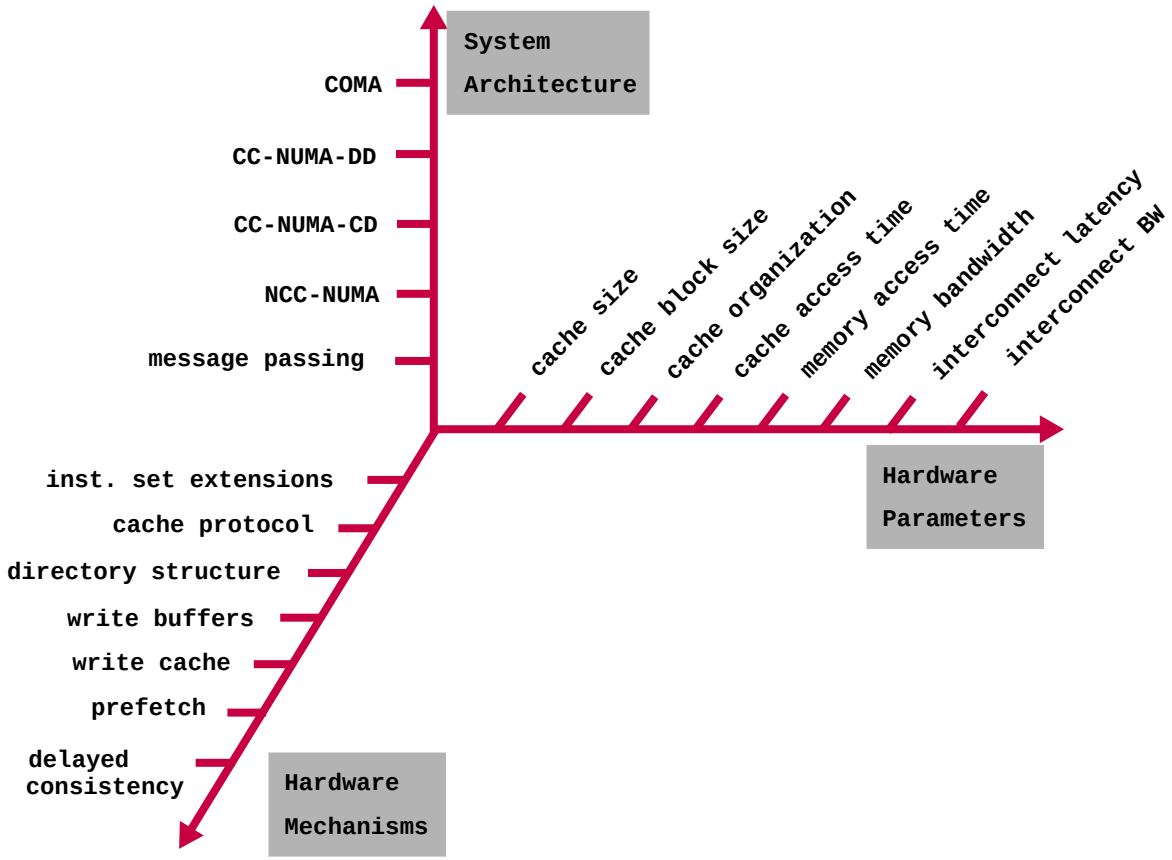


Fig. 14 illustrates the three dimensions of flexibility in RPM. Changing hardware parameters such as cache size, block size or latencies is done by modifying constants in the VHDL programs, which must then be recompiled (The only exception is the interconnect latency and bandwidth, which are adjusted at system configuration time.) Within a class of systems, hardware mechanisms, such as the cache protocol, can be changed. Usually this requires a moderate amount of reprogramming. Designing the FPGA programs for different system architectures is a major

endeavor. In the long run we hope to have all the designs stored in a database so that configuration time is limited to downloading the FPGAs in each board.

7. PERFORMANCE OF RPM

Table 2 shows the slowdown factor (which is the processing speed ratio between the target system and the emulator) for various uniprocessor technologies in the target systems. These slowdown factors can be predicted accurately in our emulation approach and they are independent of the number of processors (from 2 to 8) since the number of processors is the same in RPM and in the target.

Table 2: Slowdown factors between target and RPM

Target Uniprocessor Speed	50 MIPS	100 MIPS	200 MIPS	500 MIPS	1 GIPS
Slowdown	40	80	160	400	800
Time on RPM per Second of Target Execution	40 seconds	1 minute and 20 seconds	2 minutes and 40 seconds	6 minutes and 40 seconds	13 minutes and 20 seconds
Time on RPM per Minute of Target Execution	40 minutes	1 hour and 20 minutes	3 hours and 40 minutes	6 hours and 40 minutes	13 hours and 30 minutes

Table 2 also shows the time taken by RPM for different execution times of the target. Clearly we can reasonably expect to obtain experimental points for realistic workloads for systems with processors of up to 1 GIPS. Some of these experiments would take months to run on a current software simulator and the simulation would have to be significantly simplified and abstracted.

8. COMPARISON WITH OTHER APPROACHES

The methodologies for evaluating multiprocessor systems have slowly evolved in the past. Initially, analytical models or trace-driven simulations were used. These approaches evaluate hardware systems at a coarse level of detail. Recent breakthroughs in simulation methodology have opened the possibility of efficient, detailed and flexible evaluations.

8.1. Software Simulation

Event-driven simulators need some mechanism to schedule events, which include instruction execution and memory system operations. In an execution-driven simulator such as Tango [5], or the Wisconsin Wind Tunnel (W.W.T.) [13] each instruction is run directly on the host machine. The processors and the components of the memory systems are simulated as processes or threads, and each event requires a context switch. The source or the binary codes are instrumented to avoid scheduling an event at every instruction execution. Instrumentation code is added at basic block boundaries and at each "sensitive" data access (usually shared-data accesses). Such manipulation of the code reduces the frequency of context switches. By contrast, program-driven simulators such as Cache-Mire [2] interpret each instruction in software. Cache-Mire also relies on activity scanning (rather than an event list) for scheduling activities from processors and memory. The pay-offs are that 1) the whole simulation runs in the same context, saving expensive context switches, and 2) instrumentation of the code is not required. The performance impact of interpreting target instructions in software depends on the complexity of the memory model. If the memory system is complex, our experience with Cache-Mire shows that instruction interpretation represents less than 10% of the total simulation time.

Software simulation, whether it is execution- or program-driven, is slow. A simulator such as Tango or Cache-Mire can execute in the order of 10,000 instructions of a target multiprocessor with a complex memory model per second on a current 50 MIPS workstation. This low performance is due to multiple factors:

• the overhead due to event scheduling (e.g., context switching and event list management or activity scanning);

• the code expansion due to code instrumentation to keep track of target instruction execution times in execution-driven simulators (which was reported to be between 2 and 3 in [5]) or the overhead of decoding and executing instructions in program-driven simulations;

• the management of timestamps associated with events;

• the collection of performance data;

• the semantic gap between hardware mechanisms and their execution on the simulator (the fact that each basic activity which takes one cycle in the hardware of the target takes several instructions to simulate on the host); and

• the speedup of the target multiprocessor.

To keep simulation times reasonable, the data set sizes of the workload must be drastically reduced. Observations made on the small data set sizes must then be extrapolated to the workload with the actual data set size, a difficult task which has never really been validated.

A common drawback of all simulations is that they abstract the behavior of the target multiprocessor. Many effects are ignored or approximated, on the premise that they are negligible. In some cases some key hardware components and physical events are totally removed from the simulation. For example, it is not uncommon that a simulator avoids simulating the caches and the data transfers among them. The validity of these simplifications is usually not verified and relies on the experience and judgement of the evaluator. Besides performance, design verification is a critical issue in these complex systems; simulation and formal techniques can help but they are so abstracted that they cannot detect all design errors. Furthermore, simulators often yield little insight in the complexity of the actual implementation.

Trying to compare the efficiency of RPM with existing simulators is a hazardous task at best, because the speed of a simulation depends to a large extent on the level at which the hardware is abstracted. However, we have used two state-of-the-art software simulators, Cache-Mire [2] and TangoLite (an optimized multithreaded version of Tango [5]) to try to quantify the relative performance of the two approaches.

We have run Cache-Mire simulations of some SPLASH benchmarks on a Sun SPARCStation 10 Model 30 with 128 Mbytes of main memory and no off-chip cache. This machine is rated at about 40 SPEC MIPS (36 MHz CPU). The benchmarks we have run are: MP3D with 10K molecules, for 10 iterations, Water with 64 molecules, and Cholesky with matrix bcsstk14. In each

case we have simulated an 8-processor system with a complex memory system. The network in the simulation has constant latency and infinite bandwidth; the memory accesses are weakly ordered [4] and there are write buffers with multiple outstanding requests in both the first-level and the second-level caches [4]. The *simulation rate* of Cache-Mire is the number of cycles of the target system simulated per second. These numbers are shown in Table 3.

Table 3: Comparison between emulation on RPM and software simulation

Benchmark (Number of processors)	Number of References (Inst + Data)	Simulation Rate (Cache-Mire) (cycles/sec)	Simulation Rate (RPM) (cycles/sec)	Speedup (RPM/Cache-Mire)
MP3D (8)	18.5M	3,786	1.25M	330
Water (8)	136.5M	3,960	1.25M	315
Cholesky (8)	79.5M	3,426	1.25M	365

We have also experimented with TangoLite and the results were quite similar to the ones obtained using Cache-Mire. The TangoLite simulations were executed on a SGI Indigo workstation which is about twice as fast as the SparcStation 10 used for the Cache-Mire simulations (75 MHz MIPS 4400, 1 MByte secondary cache). Moreover, the memory model in these simulations was considerably simpler than the one used in the Cache-Mire simulations. These simplifications included no simulation of instruction fetches, no simulation of the memory/directory modules or the local busses, strong (instead of weak) ordering of memory accesses (which practically means no write buffers), and very little gathering of performance data. The simulation rate figures for TangoLite and MP3D were 7877 cyc/sec for eight processors, 3495 cyc/sec for sixteen processors and 1186 cyc/sec for thirty two processors. That makes it about a factor of two faster than the numbers for Cache-Mire in Tables 3 and 4, on a machine twice as fast.

Of course all these simulations are much more abstracted and simplified than the emulation (for instance, in the simulation, data movements are not simulated.) The level of implementation details in RPM is actually closer to a cycle-by-cycle, register-transfer level simulation. In practical cases, such simulations run at the rate of a few cycles per second and the speed up of RPM over these detailed simulators is up to one million.

8.2. Breadboard Prototypes

Multiprocessor testbeds were developed for research purposes in the 70's. However, at that time, the goal was to experiment with parallel software and the hardware was not configurable. Therefore, the experimental evidence obtained with them was valid only for the particular hardware.

Breadboard prototypes are also extensively used in industry to validate a new architecture and in academic research projects to explore architectures not pursued by industry. They are fast and faithful to the target system. One basic problem with breadboard prototypes is their cost and the relatively low amount of research information they provide besides proving one point. In an academic environment, the building of a prototype is a valuable experience for students in architecture and it keeps some sanity checks on the simulation experiments. In most projects however, most of the research results are still derived through simulations.

8.3. Hardware Emulation

The approach pursued in this project is intermediate between software simulation and prototyping. It does not replace these approaches but it complements them. Emulation is faster, more reliable and closer to the target system than software simulation. An emulator is a possible hardware implementation of the architecture. It is a fully functional machine on which any workload of the target machine can run. An emulator is also more flexible and easier to build than a breadboard prototype and, in terms of architecture research, we can expect to learn more from building an emulator.

The limitations of the current emulator are the small number of processors and the relatively low pclock rate. We were somewhat conservative in our design because of our lack of experience with FPGAs and with FPGA CAD tools. With current CAD tools and FPGA technologies the clock rate could easily be raised to 20MHz, and it is obvious from past trends that this clock rate will increase rapidly every year. Moreover the number of clocks in each pclock could also be cut in half if we used a processor with an on-chip instruction cache so that we do not have

to emulate instruction fetches. Such an emulator would emulate 5 Millions cycles of the target per second. Large speedups can be expected for bigger machines, with 128 or 256 processors. Table 4 shows the expected speed up over simulation for a 16- and a 32-processor emulator. As can be seen from the table, the speedup over simulation is superlinear. This happens because of two main factors: (1) as the number of processors increases, so does the number of caches, memory modules and other resources in the system that have to be checked at every global even in the simulator; (2) as the size of the target system grows the effect of contention for global resources is higher and adds to the overhead of both activity scanning and event-list simulation mechanisms.

Table 4: Performance of an emulator clocked at 20MHz and with 4 clocks per pclock

Benchmark (Number of processors)	Number of References (Inst + Data)	Simulation Rate (Cache-Mire) (cycles/sec)	Simulation Rate (Emulator) (cycles/sec)	Speedup (Emulator/Cache-Mire)
MP3D (16)	18.5M	1,856	5M	2,694
Water (16)	136.6M	1,868	5M	2,677
Cholesky (16)	113.8M	1,635	5M	3,058
MP3D (32)	18.7M	525	5M	9,524
Water (32)	136.6M	411	5M	12,165
Cholesky (32)	197M	402	5M	12,438

The low clock rate and the simplicity of the board-level design have facilitated the construction of RPM in an academic environment. The total time taken by the design and the construction was 15 months. Students working on the project gain valuable experiences in complex chip designs as well as in system architecture. Each system architecture mapped onto RPM is an actual system design, which a graduate student can complete in a matter of months by re-using the same hardware platform and thus concentrating on the essential part of the design. Once developed, the emulator provides the student with a valuable research tool.

With respect to flexibility, emulations on RPM are limited to machines with the overall organization of Fig. 1. However, in the future, programmable interconnect technology may be used to increase the flexibility of emulators. Like the crossbar switches used for years in commu-

nication and test systems, Field Programmable Interconnect Components (FPICs) at the chip level allow multiple input signals to be directed to multiple output pins. In the future it will be possible to build emulators with FPGAs and FPICs; these emulators will have greater flexibility to be configured for different target systems. Another limitation of the current emulator is the number of processors. There is no technical solution to this problem: To investigate larger machines, an emulator with more processors will have to be built. Finally, as for any piece of hardware, the efficiency advantage of an emulator erodes every year, as faster workstations and personal computers are introduced, whereas a software simulator capitalizes on the constant progress of commodity hardware. Nonetheless, given the current speed advantage of RPM, we expect that it will remain competitive with software simulators for at least ten years.

9. CONCLUSION

The first goal of the RPM project is to develop, demonstrate, and exploit a novel approach for the rapid prototyping of multiprocessor systems. The approach is based on hardware emulation. It relies on emerging technologies such as Field Programmable Gate Arrays (FPGAs) and advanced CAD tools (e.g., VHDL and its hardware synthesizers). The second goal is to provide a vehicle for the verification of complex multiprocessor systems. The third goal is to explore various multiprocessor models on the same, configurable hardware platform for processor speeds up to 1 GIPS (i.e., for processors which will be available during this decade).

To this end a multiprocessor emulator with eight execution processors has been built. The eight SPARC processors in RPM are clocked at 1.25 MHz, which means that RPM can emulate complex systems in full detail at the speed of 1.25 million cycles per second for systems of up to eight processors. RPM can be configured into various multiprocessor systems, including CC-NUMA (Cache-Coherent Non-Uniform Memory Access) architectures, COMAs (Cache-Only Memory Architectures), Message-Passing Systems, and Virtual Shared Memory Systems. Among these configurations many different hardware mechanisms can be implemented. RPM will be the first hardware platform on which we can compare the effectiveness of the different architectures

and their variants for realistic workloads such as operating system kernels, database systems, and realistic multitasked scientific workloads (including I/O) with the real data set sizes for which the target machine is built.

Acknowledgments. Besides the authors, several individuals have contributed to the project. In particular, we want to thank Per Stenström from Lund University (Sweden), Massoud Pedram from EE-Systems, U.S.C, and Jacqueline Chame, also from U.S.C. Through their University Program several companies helped reduce the cost of the hardware and software needed for the project. These companies are AMD, Synopsis, Viewlogic, and Xilinx. Finally, John Granacki from ISI offered the services of EZFAB, which is part of the ARPA-sponsored Systems Assembly Project.

10. Bibliography

[1] Arnold, J., Buell, D., and Davis, E., "SPLASH-2," *Proceedings of the 4th Annual Symposium on Parallel Algorithms and Architectures*, pp. 316-322, June 1992.

[2] Brorsson, M., Dahlgren F., Nilsson, H., and Stenström, P. "The CacheMire Test Bench: A Flexible and Effective Approach for Simulation of Multiprocessors", In *Proceedings of the 26th Annual Simulation Symposium*, pp. 41-49, March 1993.

[3] Catanzaro, B., *Multiprocessor System Architectures*. Prentice-Hall, 1994.

[4] Dahlgren, F., Dubois, M., and Stenström, P., "Combined Performance Gains of Simple Cache Protocol Extensions," *Proceedings of the International Symposium on Computer Architecture*, pp. 187-199, April 1994.

[5] Davis, H., Goldschmidt, S.R., and Hennessy, J.,"Multiprocessor Simulation and Tracing Using Tango," *Proceedings of the Parallel Processing Conference*, pp. II99-II107, August 1991.

[6] Dubois, M., Scheurich, C., and Briggs, F.A., "Synchronization, Coherence, and Event Ordering in Multiprocessors," *IEEE Computer*, Vol. 21, No. 2, pp. 9-21, February 1988.

[7] Fujimoto, R.M., "Parallel Discrete Event Simulation," *Communications of the ACM*, Vol. 33, No. 10, pp. 30-53, October 1990.

[8] Hagersten, E., Landin, A., and Haridi, S., "DDM -- A Cache-Only Memory Architecture," I*EEE Computer*, Vol. 25, No. 9, pp. 44-54, September 1992.

[9] Gustavson, D., “The Scalable Coherent Interface and Related Standards Projects,” *IEEE Micro*, Vol. 12, No. 1, February 1992.

[10] Kranz, D., Johnson, K., Agarwal, A., Kubiatowicz, J., and Lim, B., “Integrating Message-Passing and Shared-Memory: Early Experience,” *Proceedings of the 4th ACM Sigplan Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp.54-63, May 1993.

[11] Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.-D., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M.S., “The Stanford DASH Multiprocessor,” *IEEE Computer,* pp. 63-79, Vol. 25, No. 3, March 1992.

[12] Li, K. and Hudak, P., “Memory Coherence in Shared Virtual Memory Systems,” *ACM Transactions on Computer Systems*, Vol. 7, No. 4, pp. 321-359, November 1989.

[13] Reinhardt, S., Hill, M., Larus, J., Lebeck, A., Lewis, J., and Wood, D., “The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers,” *Proceedings of the ACM Sigmetrics Conf. on Measurements and Modeling of Computer Systems,* pp. 48-60, May 1993.

[14] Stenström, P.,”A Survey of Cache Coherence Schemes for Multiprocessors,” *IEEE Computer*, Vol. 23, No. 6, pp. 12-24, June 1990.

[15] Trimberger, S.,”A Reprogrammable Gate Array and Applications,” *Proceedings of the IEEE*, Vol. 81, No. 7, pp. 1030-1041, July 1993.

[16] Walters, S., “Reprogrammable Hardware Emulation Automates System-level ASIC Validation,” *Proceedings of WESCON*, 1990.