Multiprocessor Emulation with RPM: Early Experience

Michel Dubois, Alain Gefflaut, Jaeheon Jeong
Adrian Moga and Koray Öner

CENG Technical Report 95-23

Department of Electrical Engineering - Systems
University of Southern
Los Angeles, California 90089-2562
(213)740-4475

MULTIPROCESSOR EMULATION WITH RPM: EARLY EXPERIENCE

Michel Dubois, Alain Gefflaut*, Jaeheon Jeong, Adrian Moga, and Koray Öner

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562
(213)740-4475

*IRISA, Campus de Beaulieu
35042 Rennes Cedex
France

{dubois, gefflaut, jaeheonj, moga, oner}@paris.usc.edu

Abstract

Field-Programmable Gate Arrays is an emerging technology which promises easy hardware reconfigurability by software at low cost. Entire systems can be built in which some parts are easily programmable. Such systems are flexible hardware platforms or emulators, which are then tailored to implement various architectures. The performance of these architectures can be compared on the same hardware substrate. Besides having a large speedup advantage over software simulation, the emulator is a detailed hardware implementation of the architecture --including I/O-- on which complex software systems can be run without code instrumentation and it is a more convincing proof of concept. On the other hand it is much more cost-effective than a full-fledged prototype.

We have built a multiprocessor emulator called RPM --Rapid Prototyping engine for Multiprocessor systems. RPM can emulate various configurations of shared-memory and message-passing systems. The bandwidth and latency of various components can be easily modified to match various processor, memory and interconnect technologies. In this paper, we present the modeling methodology, the performance collection mechanism, the calibration of the emulator as well as our first results obtained for the emulator of a cache-coherent non uniform memory access multiprocessor (CC-NUMA).

Keywords: Multiprocessor, Emulation, Shared-Memory, Message-passing, Caches, COMA, CC-NUMA, Sequential Consistency, Field Programmable Gate Arrays.

1. INTRODUCTION

There are currently many competing ideas to implement multiprocessor systems. The virtues of message-passing versus shared-memory are still being debated [1] [16]. Among shared-memory systems, many variants, ranging from Virtual Shared-Memory systems [18] or networks of work-stations to cache-based multiprocessors with hardware-enforced coherence [27] are under intense scrutiny in the research community. Several projects both in industry and academia are currently evaluating new ideas in multiprocessor systems by building prototypes [1] [5] [14] [15] [16] [17] [20] [21]. In other projects, abstracted machine prototypes are developed on software simulation platforms [6] [10]. Both approaches have their advantages and shortcomings.

Hardware prototyping is extremely costly. It requires engineering skills and a technical environment not always available in an academic or even an industrial laboratory setting. A prototype verifies only one (or a few) design point and takes years to build. This takes place in the backdrop of the relentless rate of technical improvements in the computer architecture field. Processor technology is improving at a dizzying pace. New ideas in multiprocessor architectures spring up, flourish and fade away in the span of a couple of years. As a result, prototypes are usually obsolete by the time they are built or they have to keep adapting, incorporating new ideas as they come along, and the hardware is never really built. The lure of prototypes is that they are extremely realistic and attempt to exploit the current technology as much as possible. One problem that has plagued prototypes however is the fact that they are hard to observe. Even if a large number of hardware monitors are included to try to cover many possibilities there are always limits as to what can be observed, once the hardware has been frozen.

Software simulations may range from detailed cycle-by-cycle simulations of a target, which is very slow, to an abstracted simulation which can go all the way to a simple trace, in which no machine is simulated but instructions for different processors are simply interleaved and some events such as cache misses are counted. Abstracted simulations of the latter type can be very fast and yield useful information about program behavior. However the more abstracted the simulation, the less convincing it becomes. Parameters are guessed without an actual implementa-

tion, effects are neglected without really verifying the validity of the approximation, and entire parts of the machine are sometimes removed from the simulation to simplify it. There is a trade-off in software simulations between realistic simulation speeds and realism and accuracy. To solve this dilemma, more efficient techniques are being developed to speed up software simulation, including parallelization [13] [19] [24].

Hardware emulation, the approach adopted in RPM (Rapid Prototyping engine for Multiprocessors) is an intermediate approach between software simulation and hardware prototyping. RPM is a common hardware platform or substrate on top of which various multiprocessor architectures can be implemented and compared. The parameters of the hardware substrate are also variable so that various technological environments can be reproduced. The performance methodology relies on three key ideas: multi-cycle pclock emulation (each pclock --or processor clock-- of the target machine is emulated by several clocks of the emulator), time scaling (the *absolute* speeds of all components are different in the target and in the emulator but their *relative* speeds are the same) and count memory (a programmable event histogramming mechanism).

The overall architecture of RPM and the details of the hardware design can be found in [4] and [22]. This paper relates our initial experience with our first emulator, a Cache-Coherent Non-Uniform Memory Access (CC-NUMA) architecture. In the following we will briefly describe the hardware substrate of RPM and the allocation of resources for our first emulation in Section 2. Then, in Section 3, we show how the methodology has been successfully applied in a concrete case. The calibration and debugging of the emulator is a difficult task, explained in Section 4. We show the results obtained on systems of different sizes and with different application data set sizes and then conclude in Sections 5 and 6.

2. OVERVIEW OF RPM AND THE CC-NUMA EMULATION

2.1. Hardware Substrate

RPM is made of nine SPARC processors connected to a Futurebus+ backplane and is currently

clocked at 5 MHz. Normally, one processor acts as an I/O node. The single chip processors have both an integer and a floating-point pipelines and the execution of floating operations are over-lapped with the execution of integer instructions. They have no on-chip cache and therefore all instruction fetches and data accesses are observable. The block diagram of each processor board is shown in Figure 1

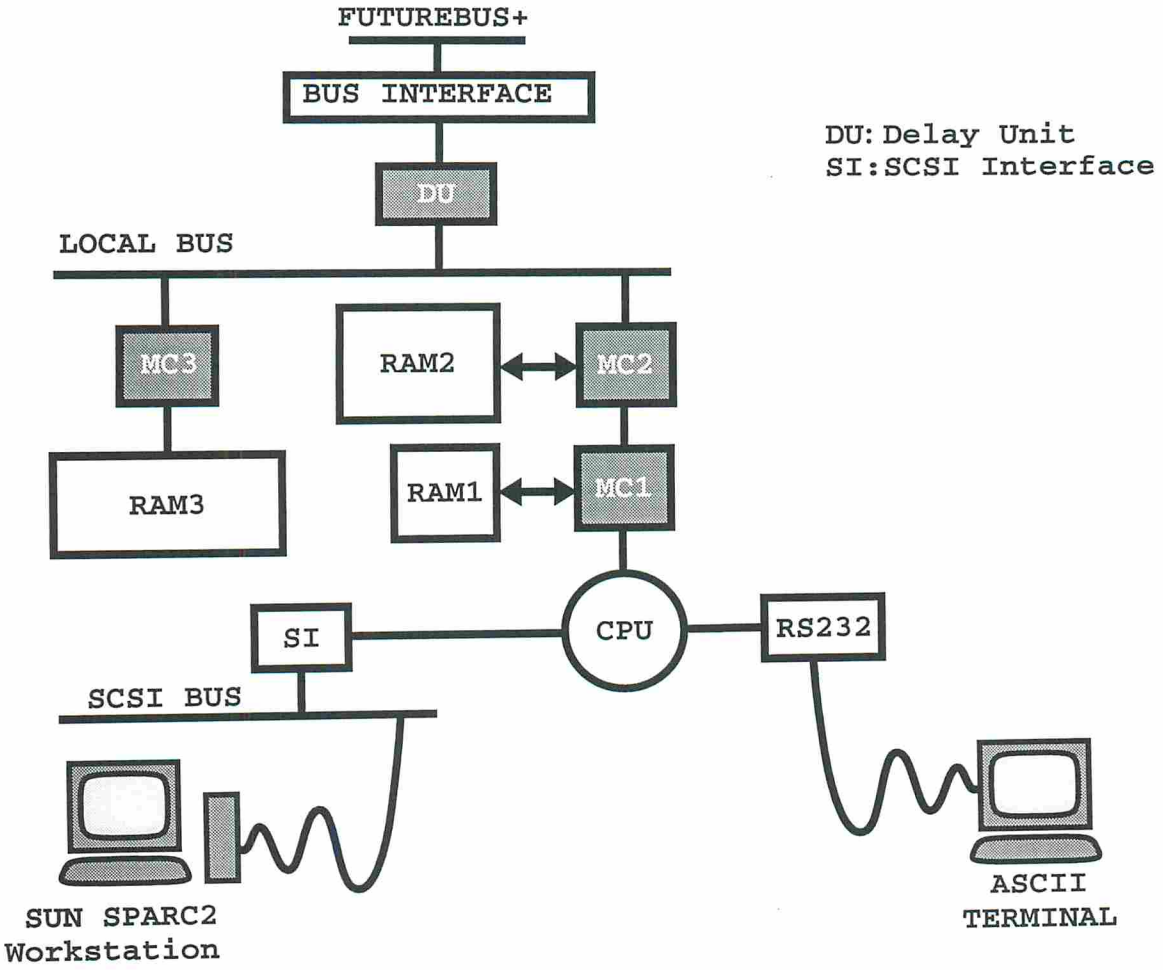

Figure 1 Block Diagram of RPM's Processor Node [4]

The CPU is an LSI Logic L64831 SPARC IU/FPU. The first-level cache is built with 2 Xilinx 4013 FPGAs (MC1) connected to 2 Mbytes of SRAM SIMMs (RAM1); the second-level cache is built with 2 Xilinx 4013 FPGAs (MC2) connected to 8Mbytes of SRAM (RAM2) and the main memory (RAM3) is made of 64Mbytes of DRAM. The main memory controller is made of 2 Xilinx 4013 FPGAs plus an off-the-shelf DRAM controller (Cypress CYM7232). The delay unit 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.The FutureBus+ chip set comes from Newbridge and National Semiconductors and it includes bus transceivers plus the Newbridge LIFE chip. The SCSI interface of the I/O board is attached to the SCSI bus of a SPARC station II which currently serves as an I/O server and console for RPM. The RS232 serial interface can connect to a terminal and is used mostly for debugging purposes. The board size is 22"x16".

Each processor board has a 2 Mbyte first-level memory (SRAM), an 8 Mbyte second-level memory (SRAM) and 64 Mbytes of main memory (DRAM). In a typical emulation, a fraction of these memories emulates the target system's caches and memories and the rest is used for performance collection and for the emulation of special registers and buffers (such as a T.L.B.).

The first-level memory controller drives the emulation of each pclock. Typically, at the start of each pclock emulation, it resumes the processor execution for one cycle, receive the next processor access (if any), blocks the processor, and then emulates the memory access (in one or several pclocks). The number of clocks per pclock can be easily changed and it depends on the complexity of the pclock emulation.

Each controller is implemented with two FPGAs (Field Programmable Gate Arrays). The emulation of each pclock is implemented by a combination of control in the FPGAs and memory space in the RAMs attached to the memory controllers. The Delay Unit (DU) is a programmable chip which emulates variable interconnection delays. Messages that are sent out of the processor node are stored in the Delay Unit for an amount of time that is programmable. If L is the length of a packet, then the delay is equal to $\alpha + \beta \times L$, where α is a fixed delay per packet and β is the time per 32-bit word transfer. The FutureBus+ interconnection is 32-bit wide, and transfers one 32-bit word every 100nsec, for a total bandwidth of 40 Mbytes per second. The main memory speed can be modified by suspending memory requests using interleaving registers as described in [4]: after the memory controller has received and decoded a request packet, it suspends the request for a programmable time; it then resumes the request and sends out the messages resulting from the transaction. During this entire time, the memory appears busy. The SCSI interface and serial port allow every board to be configured as an I/O node.

Memories of all boards can be accessed in *Test mode* or in *Emulation mode*. In Test mode (mostly used for booting and debugging), the memories on all boards are all part of a contiguous address space and every location can be accessed randomly. To configure RPM into a target multiprocessor, the FPGA programs of all three controllers are written in VHDL, compiled and then

downloaded through the I/O node as part of the booting procedure. Once the FGPAs are programmed, the machine is in test mode. Memories are initialized, the code and data of the application are downloaded into RPM's main memory and the machine then switches to emulation mode where memory accesses are restricted by the addressing mechanisms of the target machine.

2.2. An Emulator for Small-Scale CC-NUMAs

Today's successful multiprocessors are mostly small-scale systems (2 to 16 processors) with shared memory and cache coherence enforced by a snooping protocol on one or multiple buses. However, as the speed of uniprocessors keeps increasing it becomes more and more difficult to support more than a few processors on buses. As an alternative to snooping, industry and academia are exploring other, point-to-point interconnect such as rings [3] or crossbars to connect ultra-fast processors in a small number (1 to 32). Our first emulator is a CC-NUMA under strong ordering of shared-memory accesses [11]. Strong order is still a requirement of many commercial systems. For example, even if the SPARC V-9 (RMO) memory model is very weak, the current version of Solaris 2.4 still assumes a strong memory order (close to sequential consistency) and disables latency-hiding hardware optimizations [7].

2.2.1. Protocol

The protocol is write-invalidate with a directory organization based on Censier and Feautrier's design [8]. Besides the presence bits and the modify bit per block, we have added some transient state bits to support concurrent transactions on different directory entries. A miss request first goes to the home memory. Before returning a copy of the block, the home memory must sometimes invalidate copies (write miss in the presence of multiple remote copies) or obtain the dirty copy from the owner (miss in the presence of a dirty remote copy). This latter transaction takes four traversals of the interconnect (requester-to-home, home-to-dirty, dirty-to-home and home-to-requester). After invalidations are sent by the memory controller the block state is set to transient and the presence bits indicate which copies have pending invalidations. While invalidations are

propagated, the memory controller is released and any other request to the block is *nacked* by the home node. As invalidations are acknowledged, the memory controller resets the presence bits; once all acknowledgements have been received the controller completes the transaction and sets the block to a stable state. More details on this protocol can be found in [23].

Resource	Event	Basic Hardware (clocks)	100MHz processors (pclocks)
FLC	Data Read (single) or Instruction Hit	8	1
	Data Write Hit (single)	24	3
	Data Read Hit (double)	16	2
	Data Write Hit (double)	32	4
	Fill from SLC	16	2
SLC	Hit from FLC	40	5
	Hit from Bus Interface	24	3
	Miss Detection	16	2
	SLC fill	24	3
	SLC restart	24	3
Internal Bus	Request Packet	4	1/2
	Data Packet	8	1
DU and FutureBus+	Request Packet	24	8
	Data Packet	24	20
Memory	Miss (T_{miss})	40	15
	Send k Invalidations (T_{inv})	(10+4k)	(1+2k)
	Ack Invalidation (T_{ack})	24	5
	Receive Write Back (T_{wb})	40	15
	Get Block from Dirty (T_{md})	56	8
	Send Dirty Block to Requester (T_{mr})	72	9
	Nack (T_{nack})	24	5

Table 1 RPM Hardware Latencies (no conflict)

Miss is the time to receive a packet, fetch the block from DRAM, update the directory and send a packet back to the requester. *Send k Invalidations* is the time to send k invalidation packets. *Ack Invalidations* is the time to receive an acknowledgement for one invalidation and update the directory. *Receive Write Back* is the time to receive a block packet, store the block in DRAM, and possibly forward the block to the requester. *Get Block from Dirty* is the time to receive a packet from the requester, check the directory, and send a packet to the dirty node. *Send Dirty Block to Requester* is the time to receive the block copy from the dirty node and send it to the requester with ownership (memory is not updated). *Nack* is the time to receive a packet, and negatively acknowledge it.

2.2.2. Architectural Parameters

The first-level cache is a 64kbyte direct-mapped write-through cache with a block size of 16 bytes. The second-level cache is a 1Mbyte direct-mapped write-back cache with 16 byte blocks. The emulated memory size on each board is 32Mbytes. The basic hardware latencies listed in Table 1 (second column) are the number of cycles needed for the emulation. They were obtained by counting cycles in the RPM hardware. They are the lowest possible latencies for the CC-NUMA emulation.

The latencies of second level cache read misses comprises three terms: on-board delays, memory delay, interconnect delays. From Table 1 the following equations are applicable for the latencies (in clocks) of read misses in the absence of conflicts.

- SLC read miss from Local Home Memory: $36 + T_{miss}$
- SLC read miss from Remote Home Memory (2 hops): $48 + T_{miss} + 2 \times (24 + \alpha) + \beta \times 4$
- SLC read miss from Dirty Node (4 hops): $96 + T_{md} + T_{wb} + 4 \times (24 + \alpha) + \beta \times 8$

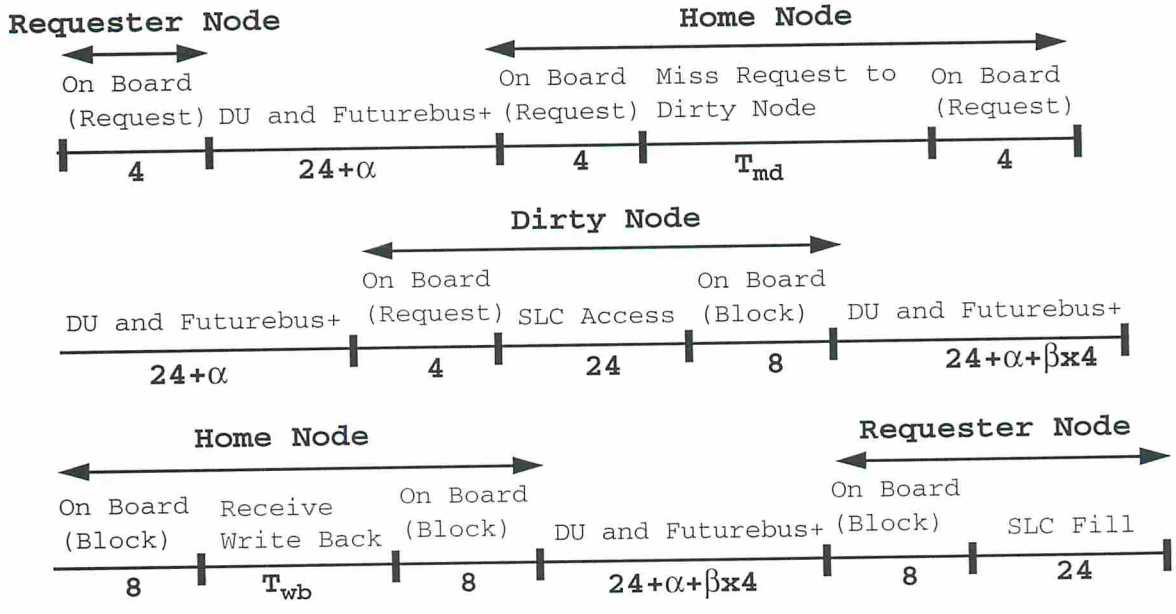

Figure 2 Timings of a Read Miss with Fill from a Dirty Node (no conflict)

Figure 2 shows the decomposition of the latencies of a read miss on a dirty copy. The basic hardware latencies for read misses (α and $\beta = 0$) are 76 clocks (local), 136 clocks (home), and 288 clocks (dirty). There are small fluctuations in these latencies because some timings (such as accesses to the DRAM, FutureBus+, and the second level cache) may vary slightly. However, these simple estimates are suprisingly accurate as we will see in section 4.

2.2.3. Software Environment

We have developed a software environment for RPM to run applications using the ANL programming macros to express and manage parallelism. In particular, the SPLASH and SPLASH-2 sets of benchmarks [29] are written in such a way. The original SPLASH benchmarks employ the UNIX FORK model, in which parallel processes communicate exclusively through variables that are explicitely allocated in globally-shared memory. Our emulator provides a natural support for this model by running the master process on the I/O board and slave processes on the others. The addressing space of each process consists of a private space, hosting local data, bss, stack and local heap and a shared space for the executable code and the global heap. The emulator provides access to other boards' private locations through special alternate space transfers, thus supporting the forking mechanism. Synchronization between parallel processes is based on shared-memory locks and the atomic *testandset* operation. A hardware barrier is also available in the emulator, but is only used by system routines. Some SPLASH-2 benchmarks are written for the SPROC model, in which all processes (now threads) share the same addressing space. In particular, threads can communicate through named global variables, stored in the data segment. We made minor modifications to these applications and retrofitted them to the FORK model.

As shown in Figure 3, applications for the emulator are compiled using the standard procedures and UNIX tools for a SUN-4 workstation. First, most of the library functions are statically linked directly from SUN's standard C and math libraries. Some of the functions (e.g. *sbrk(), getpagesize(), _exit(), abort(), setjmp()*) had to be rewritten, however. They are part of libcx.a. System-specific functions, such as device access and hardware barrier synchronization

are provided by the librpm.a library.

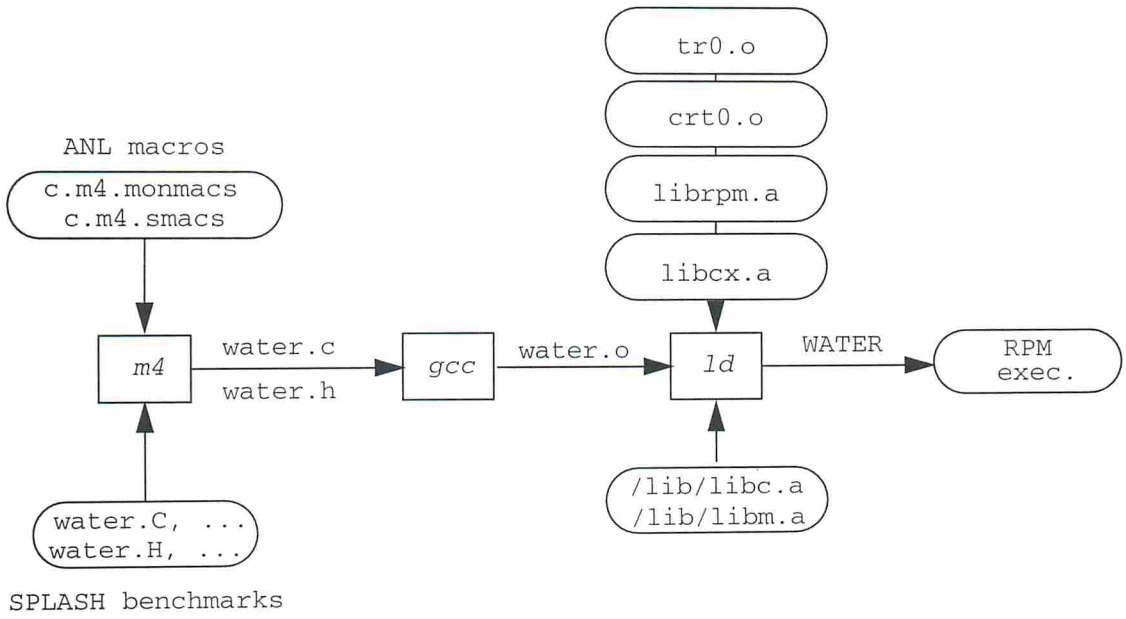

Figure 3 Software Environment to Run the SPLASH Benchmarks on RPM

Second, we had to provide system support for I/O functions. A special operating system trap handler (tr0.o) was written to support this task. It intercepts the system call parameters and passing them over a communication line to a server running on the SUN workstation acting as the emulator's console. The server performs the request and cooperates with the system call stub running on the emulator's I/O board for possible data transfers. Upon completion, the server passes a return code to the stub, which is delivered to the application as a result of the system call. The whole mechanism is transparent to the application and the appearence of a UNIX kernel is entire.

Third, a new C runtime library (crt0.o) had to be developed to take care of the initialization and termination of processes.

Once compiled, applications can be downloaded into the shared-memory of the emulator at a fixed address. To start them, control is passed from the emulator's monitor to the application entry point, after switching from test mode to emulation mode. The slave boards block immediately, awaiting commands from the master board. After some initialization, including relocation

of data into the private space, the master board proceeds through the sequential part of the application. Eventually it forks a number of slave processes and they all stop at a hardware barrier to start collecting statistics before beginning the parallel section. A similar barrier is entered upon termination of a process and then execution returns to the monitor routine.

3. PERFORMANCE METHODOLOGY

3.1. Multi-cycle Pclock Emulation

Each pclock of the CC-NUMA emulator is executed in eight clocks. The corresponding reduction in emulation speed is compensated by three advantages. First we can emulate complex mechanisms without sacrificing flexibility. Second, the latencies of the basic hardware are independent of the latencies of the target; even if a large number of clocks are needed to move packets on the board and on the bus, and to emulate memories and complex directories, the target may still have very low latencies when expressed in pclocks. Finally, the bandwidth of the FutureBus+ is not a constraint in RPM. If processors executed at the rate of one clock per pclock, the bus would be a serious bottleneck.

3.2. Time Scaling

3.2.1. Latencies

In RPM, the latencies of main memory and of the interconnect can be easily changed. We can emulate systems with various processor, memory and interconnect technologies. The simple rule is that latencies measured in pclocks must be the same in the target and in the emulator. For example, if, in the target system, processors are clocked at 100MHz (pclock=10nsec), interconnect delay is 80nsec and DRAM access time is 100nsec, then the interconnect delay must be 8 pclocks (or 64 clocks) and each DRAM access must take 10 pclocks (or 80 clocks) in RPM.

The third column of Table 1 shows the latencies for a target systems with 100MHz processors and 100nsec DRAM access times. The values for memory/directory accesses assume that the

directory is built in fast SRAMs, that the DRAMs can fetch an entire block in one cycle, and that the memory controller can be clocked at the speed of the processor and works in parallel with the DRAM access. The values of 8 pclocks and 20 pclocks (obtained by setting α = 40 clocks and β = 24 clocks) for the latencies of the target interconnect were chosen from the hypernode of the Convex Examplar, which is an 8 100MHz processor cluster connected by a crossbar switch [9].

From the numbers in Table 1, and the expressions for the read miss latencies in the target, the number of pclocks to service a SLC read miss in the CC-NUMA emulation is 19.5 pclocks if serviced by the home locally, 49 pclocks if serviced by the home remotely and 91 pclocks if serviced by a dirty node.

3.2.2. Bandwidths

Besides latency, bandwidth is also a critical performance parameter for any hardware resource. Many protocol mechanisms such as updates and prefetches work wonders when memories and interconnects are modelled with infinite bandwidth, but perform poorly in a real system because of the limited bandwidth [12].

The bandwidth of the memories and directories in our CC-NUMA emulator is directly related to their access latencies since we assume that memory on each board is not interleaved. However, we have included a mechanism in the RPM hardware to emulate memory interleaving (up to 8-way on each board) and thus increase the memory bandwidth for given memory access latencies. This mechanism is based on a set of interleaving registers and was described in [4].

The current width of the FutureBus+ is 32-bits (extensible to 64 bits) and its bandwidth is 8 bytes per clock or 64 bytes per pclock. With a target pclock rate of 100MHz, this corresponds to an interconnect bandwidth of about 6 Gbytes per second in the target system. This is a huge, practically infinite bandwidth which increases proportionnaly to the target's pclock rate and this explains why we have observed very low interconnect traffic in our experiments (see section 5). The FutureBus+ bandwidth can be modified by changing the bus clock rate or the bus width and

by padding packets with extra bytes. It is not difficult to hold the bus for longer than necessary on each bus transfer, in effect adding dummy bytes to packets as they reach the FutureBus+ interface.

3.3. Count Memory

The primary two mechanisms to collect performance statistics in RPM are memory-mapped counters (implemented in the FPGAs) and memory-mapped event-count memory (implemented in memory). Memory-mapped counters are simple counters that can be read/written as any memory location. Such counters can measure the total execution time, the utilization of the processors and the controllers, and can count events. So far, the only counter we have implemented in the FPGAs is a 40-bit pclock counter in the first-level cache controller of every processor to record the total execution time of each processor.

Event-count memories consist of a set of counters implemented in the SRAMs of the caches and the DRAM of main memory. Events are mapped to memory addresses, and, at the occurrence of a given event, the value stored at the corresponding memory location is incremented. Therefore, to implement event-count memories, it is necessary to have some extra hardware to read a counter from count memory, increment its value and write the updated value back to memory. Only one count-memory location can be accessed in each pclock

We associate a 'basic' event to each bit of an address in count memory. For example, one address bit may be associated with the event *hit/miss*, one address bit may be associated with the event *read/write* and one address bit may be associated with the event *instruction/data*. At the occurrence of a data read miss, the signal *hit/miss* indicates a miss, the signal *read/write* indicates a write and the signal *instruction/data* indicates a data access and the corresponding count memory location is incremented. Consequently the total number of 'data-read-misses' is accumulated at the memory location addressed by these three basic events (data, read and miss); the total number of data-reads is obtained by adding the contents of the locations 'data-read-misses' and 'data-

read-hits', at the end of the emulation.

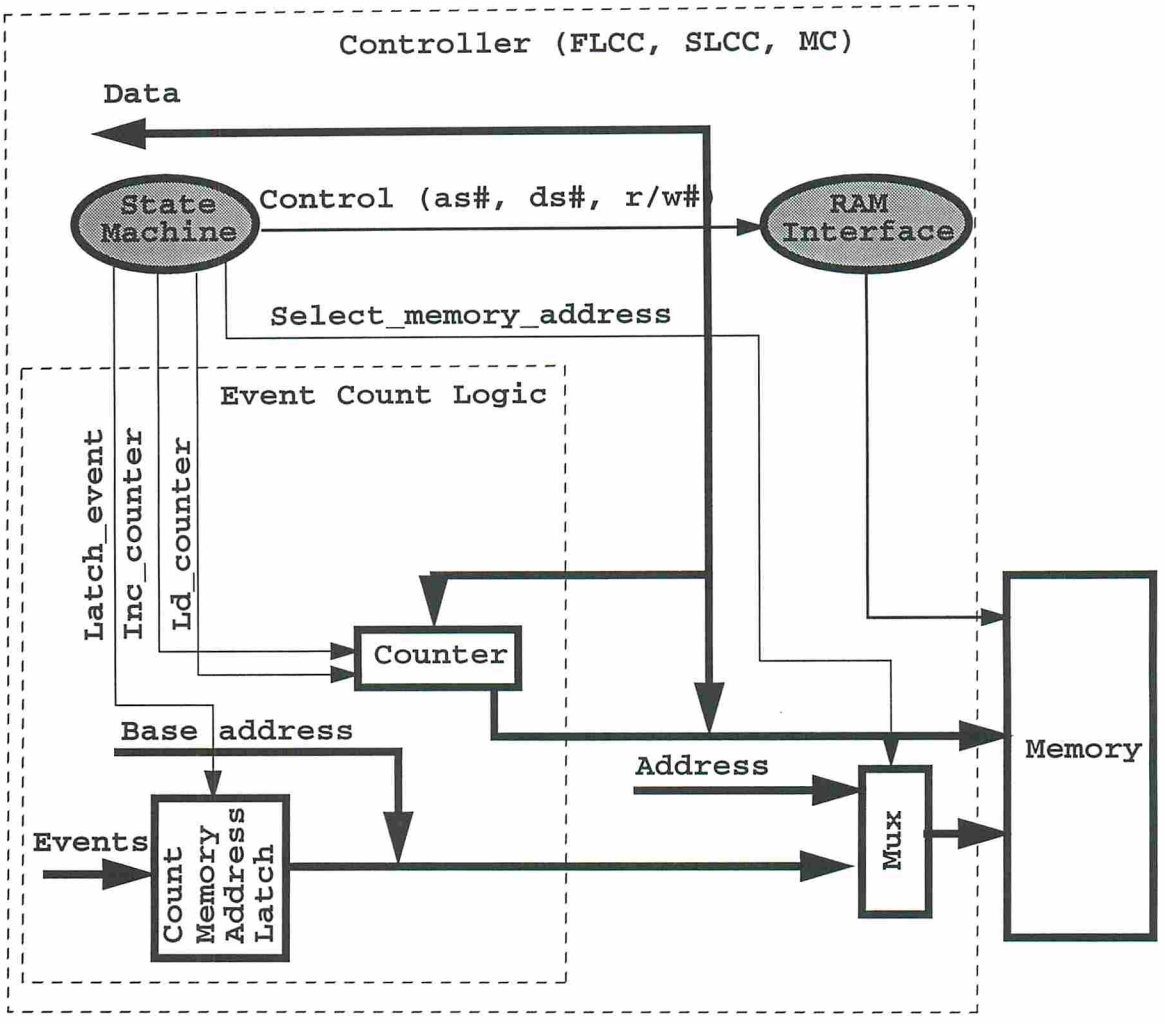

Figure 4 Control for Count Memory

This logic is present in each cache controller and in the main memory controller. The memory address is provided either by the controller or by the event count logic. The event count logic stores the address of the event (or combination of events) occurring in the current cycle in the address latch; this address, concatenated with the base address of the count memory, accesses the memory during the performance cycles. The value in the memory location is loaded into a counter; the counter is incremented and the value is stored back into the same memory location.

In the first-level cache, an event counter is incremented at each pclock, as part of the pclock emulation, whereas, in the controllers of the second-level cache and of the main memory, an event counter is incremented on each request to the controller. The hardware in each controller to implement count memory is shown in Figure 4; Figure 5 displays the composition of addresses to the event counters in the three memories, obtained by merging hardware signals corresponding to

various occurrences and resource states.

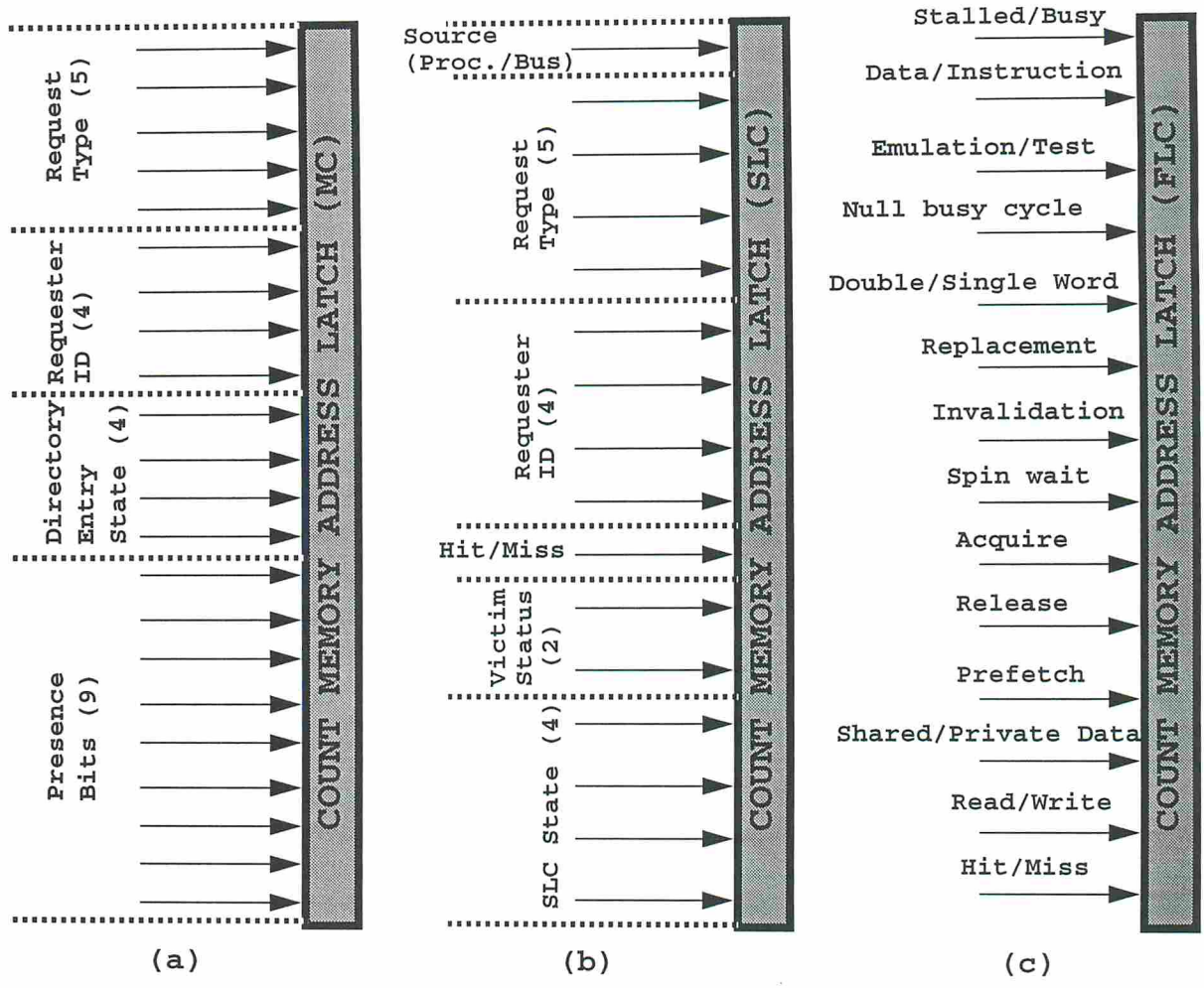

Figure 5 Addresses Used to Access Count Memory in MC, SLC and FLC

The main memory contains 4 M 32-bit counters (a); for each request type, each request source, and each global state of the block, there are 512 counters, one for every possible distribution of the copies among the processors. The second-level cache count memory contains 64K 32-bit counters (b); the location of the event counter depends on the request type, the source of the request (bus or local processor plus the ID of the processor), the state of the block in the cache, the state of the replaced block and the value of the hit/miss line. Finally the count memory of the first-level cache contains 16 K 32-bit counters (c), detecting various data or instruction cycles in the processor.

Each processor board has three sets of event counters. At the end of an emulation run, all processors store their counters in their main memory in parallel; then the I/O processor uploads all counters to the SUN host, where the events are summed up and combined to obtain the counts of meaningful aggregate events.

4. VERIFICATION, CALIBRATION AND PERFORMACE DEBUGGING

Once an architecture emulator is built, it is very important to verify that the characteristics of the architecture, such as cache sizes, cache organizations, and access times, are correct according to the original specifications defined for the target system. For an emulator built on top of RPM, it is also necessary to verify that the performance counters recorded during executions are accurate. In this section we present the three steps used to calibrate and verify the correctness of the CC-NUMA emulator. In a first step we verify the performance counters and access latencies using a set of small programs; each program repeats a specific and predictable memory access pattern. In a second step, we verify the architecture characteristics of the emulator using a method derived from [25]. Finally, we verify the emulator by comparing results obtained with a simulator and the results obtained with the emulator running some of the Splash-2 applications.

4.1. Checking Performance Counters and Latencies

To test and verify the count-memory mechanism, we have developed a set of simple programs, each of which is a simple loop exercising a particular memory access pattern. The statistic collection is performed only during the execution of the loop. As the behavior of each of these loops is perfectly known, the value reported by the counters can be predicted and then verified.

Figure 6 gives a simple example of a loop where each read access misses in the first-level cache but hits in the second-level cache. In a first step, the second level cache is filled with data by reading an array. The main loop is then executed and generates a miss for each read access. With such a simple program we were able to verify the counters for the number of read accesses in the FLC, the number of read misses in the FLC, the number of read accesses received in the second-level cache and the number of read hits in the second-level cache. More than 20 of these small programs were used to address all the possible situations that can happen during the execution of a real program

Additionally, this approach has been used to verify the timing characteristics of the emulator. Consider the previous example. The number of pclocks recorded in read misses divided by

the global number of read misses gives a good approximation of the time to service a read miss in the second-level cache..

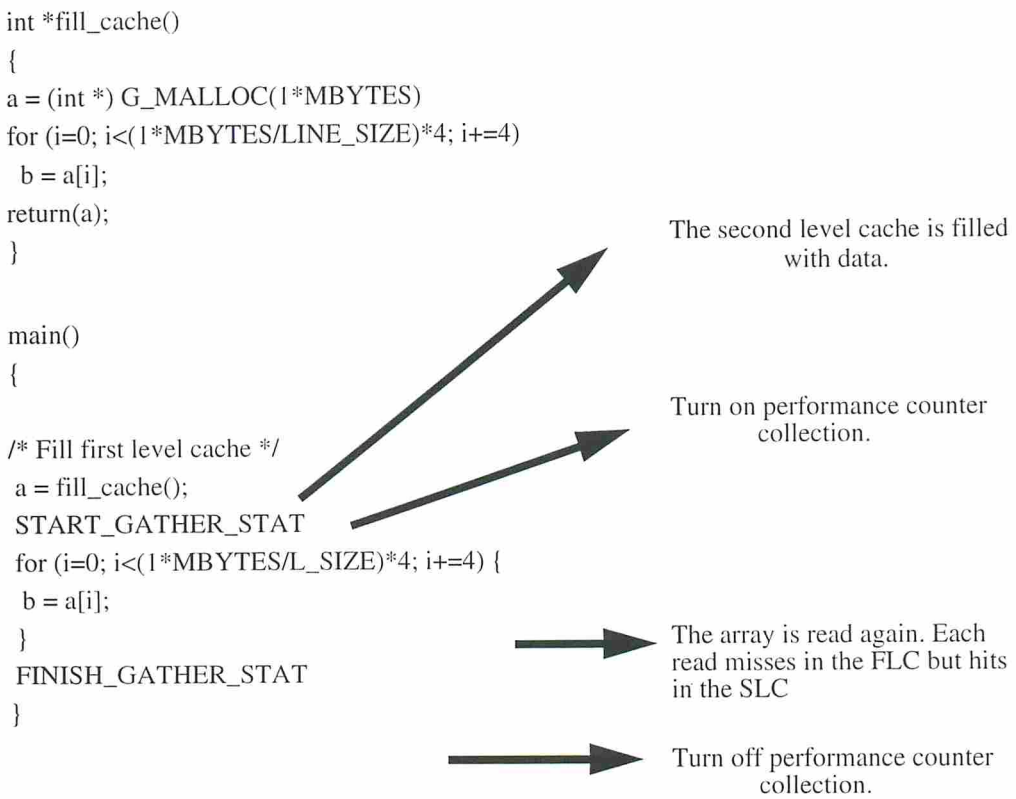

Figure 6 Example of one of the Programs Used to Verify the Performance Counters

This strategy has been extremely valuable to identify and correct a large number of bugs in the performance collection, in particular in the first-level cache where a counter is updated in each pclock and where a simple error in the address of a counter leads to incorrect results. As an example, an inconsistency was discovered between the number of misses issued from first-level cache and the number of requests received by the second-level cache for double read or write accesses. To fix this bug, we had to add new counters for double accesses.

4.2. Verifying the Architecture Parameters

To verify that the emulator respects the specifications of the target architecture, Saavedra's technique described in [25] and based on micro-benchmarks has been applied to the emulator.

The micro-benchmark is made of multiple loops. In each loop, the processor reads every S-th (Stride) element of an array of size R. After each iteration, the stride is multiplied by 2 until it reaches R/2. When S reaches R/2, R is increased and S is reset to its initial value. For each value of R and S, we first run the micro-benchmark with the read access, and then we run it again with the read access replaced by a noop. The times are measured using the pclock counter provided by the RPM hardware in the first-level cache controller. The difference between the execution times of these two loops divided by the number of iterations yields the average read access time. A diagram representing the average time to read a single element for different values of R and S, is obtained. On the diagram, each curve corresponds to a value of R. The same benchmark with write accesses could be used to measure the timings of write accesses.

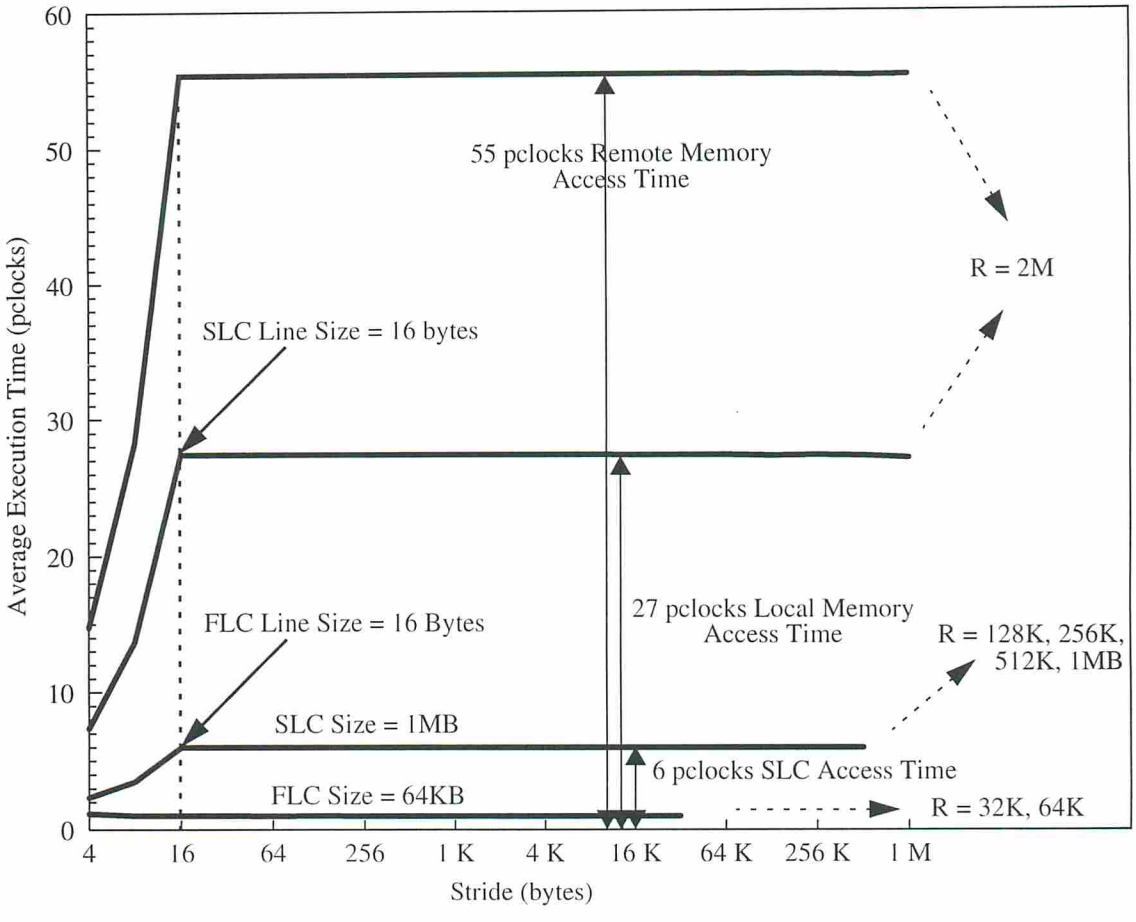

Figure 7 Performance profile of the CC-NUMA emulator

Different regimes are observed on a graph resulting from the execution of a micro-benchmark. For example, when the size of R is inferior to the size of the first-level cache, all elements accessed in the loop fit in the first-level cache and the average read time as a function of S is a constant, as shown by the bottom curve of Figure 7. By varying the values of S and R, other regimes where the read access hits or misses in the second-level cache are observed. These different regimes allow us to observe experimentally the cache size, organization, and latency times through various features of the diagrams, as pointed out in the figure. A more detail description of these regimes and of the way to interpret the graphs can be found in [25].

Figure 7 shows the graph obtained for the CC-NUMA emulator. From this figure, we can infer that the first-level cache is direct-mapped and that its size is 64Kbytes; we can also verify that the second-level cache is direct mapped with a size of 1MByte and that the lines in both caches are 16 bytes. Access times are confirmed as follows. A read hit to first level cache takes 1 pclock. A read miss in the first-level cache that hits in the second level cache takes 6 pclocks. A read miss in the first- and second-evel caches takes 27 pclocks if it is serviced from the local home memory and 55 pclocks if it is serviced from a remote home memory. These experimental timings are more accurate than the timings estimated by counting cycles, as we did in section 2. However, the two sets of estimates are in close agreement. Adding the first-level cache access time, the second-level cache miss detection time, the second-level cache restart time and the first-level cache fill time (a total of 8 pclocks), to the second-level cache miss times computed in section 3.2, we find a total of 27.5 pclocks (vs. 27) for a read access with a local second-level cache miss and 57 pclocks (vs. 55.5) for a read access with a remote second-level cache miss.

The technique employed in this cache parameter verification is applicable to set-associative caches as well [25].

4.3. RPMsim

RPMsim is an approximate simulator of the CC-NUMA emulator. Luiz Barroso wrote this simu-

lator in C using the CSIM package [26] and the Cache-Mire SPARC interpreter [6]. It consists of several modules implementing the processor, first- and second-evel caches, memory controller, main memory, and the shared bus.

The processor module is a simulator for the instruction set of the SPARC architecture. It has no detailed simulation of execution pipelines and it issues a memory cycle upon every invocation, including instruction fetches. The fact that we do not simulate the pipeline introduces some error in the simulation, because, in the real machine, instructions do not always complete in one processor cycle. Moreover, some cycles are null cycles. Null cycles are mostly due to annuled instructions in branch delay slots. Moreover, none of the extended precision instruction and a few double precision instructions are implemented in the simulator, as they do not appear in the code generated by the GCC compiler. However, some of them may occur in math library functions linked together with the application. To handle the situation, these functions have been implemented as application traps to the simulator. The benefit is a shorter simulation time to the detriment of accuracy. Because these functions make heavy use of registers and have few memory accesses, the caching behavior of the application is not severely affected, but the application execution time may be affected.

The timing parameters of the simulated system are embedded in the cache, memory controller and bus modules. These parameters attempt to closely approximate the values of the emulated architecture, but there are several limitations and simplifications. Double write accesses are simulated as two individual back-to-back accesses and incur the latency of a write-through twice. In the real machine, double writes are treated differently because the tag matching in the second-level cache is done just once, hence the overall latency is smaller. More importantly, the simulated clock is the processor clock and all the simulated delays have fixed, average values. In the emulator, the system clock has a frequency eight times larger and some events and arbitrations do not happen synchronously with the processor clock. Hence latencies have small fluctuations in the real case. Other minor inaccuracies are introduced in the simulation by the absence of main

memory refresh cycles in the operation of the memory controller and of other particularities of its hardware implementation.

We will report on our simulation experiments at the end of section 5.

5. Emulation Results

In this section, we present some of the performance data collected on the CC-NUMA emulator and do some preliminary comparisons with simulation.

5.1. Emulation Speed

RPM currently emulates the execution of 625,000 target pclocks per second at the 5 MHz clock rate. Assuming that every instruction executes in one pclock this corresponds to a peak emulation rate of 625,000 instructions per second and per RPM board. The emulation rate of RPM is strongly affected by the characteristics of the programs it executes. RPM approaches its peak emulation rate for programs with little memory activity and with very high computation-to-communication ratios.

As more processors are added in RPM, the effective emulation rate increases; an 8-processor RPM's emulation reaches a peak emulation rate of 5 million instructions per second. Figure 8 (b) shows how the emulation rate of RPM is affected by increasing the number of processors. All programs except MP3D are taken from the SPLASH-2 benchmark suite [29]. The selected data sets are larger than the default values for Cholesky and FFT and smaller for Barnes-Hut, FMM, Radiosity, Radix, Raytrace, and Volrend. The best emulation rate is achieved for Volrend in which RPM reaches nearly 3 million emulated instructions per second which is 60% of RPM's peak emulation rate and the worst case is for MP3D with 833,000 emulated instructions per second.

We observe that the programs with high algorithmic speedups, such as Volrend, Watersquared, Raytrace, Barnes-Hut, Radix perform very well on RPM too. By contrast, programs

with low speedups such as MP3D and FFT, perform poorly.

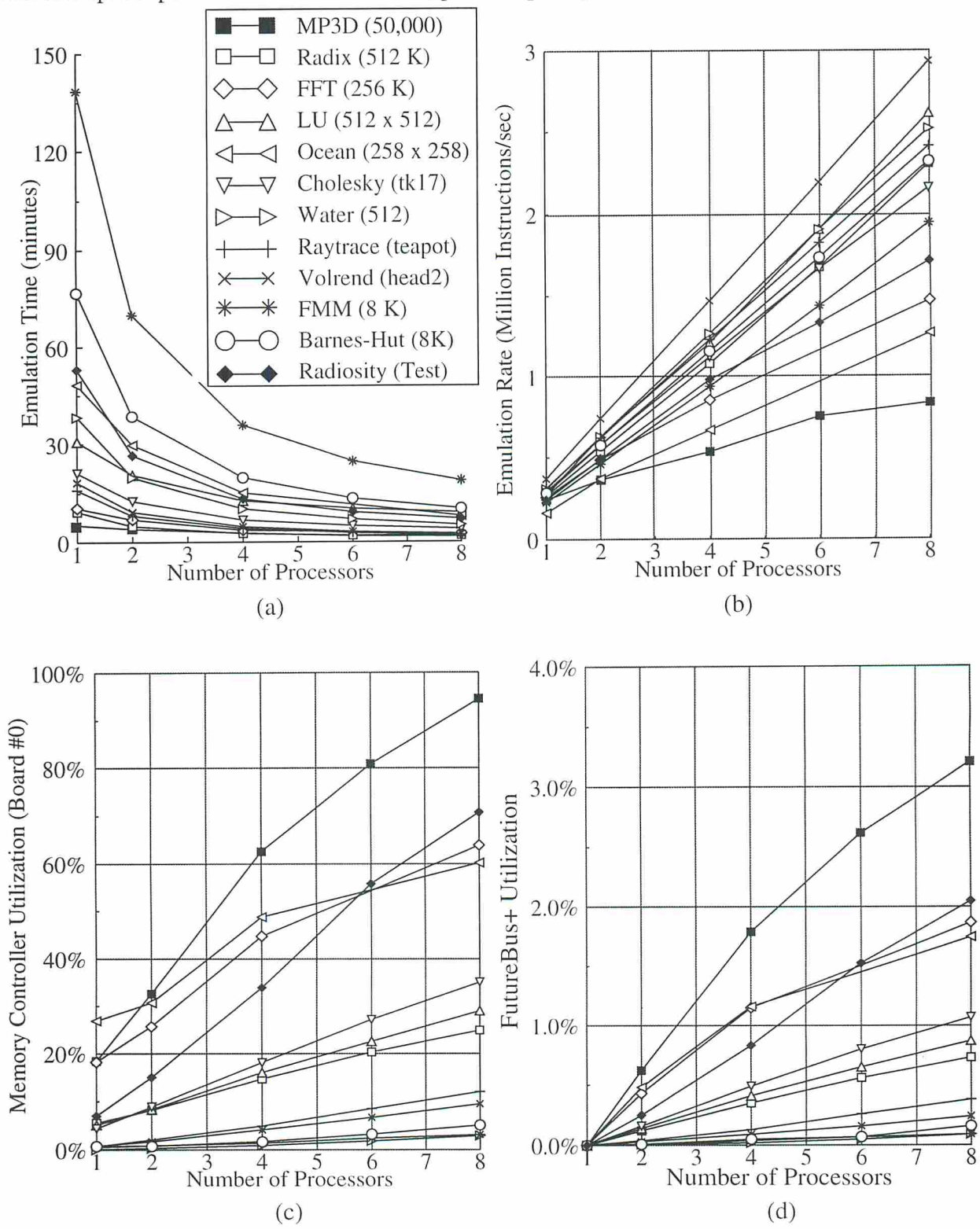

Figure 8 CC-NUMA Emulation Performance

(a) Emulation Times; (b) Emulation rates; (c) Worst-case Memory Controller Utilization (board 0); (d) FutureBus+ Utilization.

There are some exceptions. For example, FMM has a very high speedup, but its emulation rate on RPM is not that good. This is because of the huge number of local double write operations FMM executes. Similar to FMM Radiosity also has a good speedup, but mediocre emulation speed. Basically, Radiosity spends on the average 18% of its execution time on instruction stalls and 19% of its execution on data read stalls.In the current implementation of RPM local data and code are stored in consecutive boards, starting with board 0. For programs with very small data set sizes, this causes two problems: 1) the memory controller on board 0 becomes a hot spot and reaches very high utilization values, and 2) programs with very high local data and instruction misses will have very high instruction and data (read/write) stall times. In Radiosity we observe both problems: the utilization of the memory controller on board 0 reaches 70% and the number of instruction and data read misses is high. Although LU has normally worse speedup than FFT, it achieves a very good emulation rate. The main reason for this is that the work is not uniformly distributed among processors in LU. When a processor is waiting for other processors (busy waiting), it stays in a tight loop with a single data access, which hits most of the time. During this time, RPM is close to its peak emulation rate.

Ocean has normally better speedup than FFT, but its emulation rate is lower than FFTs'. Ocean exhibits a large local data traffic [29] and, in the current configuration of RPM, the local data of boards other than board 0 are remote. We can also see from Figure 8 (c) that the utilization of the memory controller on board 0 reaches 60% in Ocean. By contrast, Cholesky has lower speedup, but still achieves better emulation rate on RPM. This is mostly because of Cholesky's smaller data set size and lower data traffic.

Figure 8 (a) shows the emulation times of all benchmarks. The emulation times vary from 10 to 140 minutes on one RPM processor and from 2 to 20 minutes on eight processors. FMM takes the most time and Barnes-Hut comes second. This graph gives a feel for the speed of RPM.

Figure 8 (c) displays the utilizations of the memory controller on board 0. The memory controller utilizations are calculated in the following way. We can divide all messages that are

received and send by a memory controller into five types: 1) read miss requests, 2) write miss requests, 3) ownership requests, 4) write backs due to replacement, and 5) negative acknowledgment. Message types 4 and 5 are consumed by the memory controller and they do not generate any further messages, but message types 1, 2, 3 may generate secondary messages if the block is shared or if it is dirty in another processor cache. To find the utilization of the memory controller we first calculate the busy time of the memory controller by summing up the times the memory controller takes to process specific request types including the secondary messages and their replies. For example, the total time a memory controller spends processing read misses is calculated in the following way:

$$Nb\_read\_miss\_on\_clean * T_{miss} + Nb\_read\_mis\_on\_dirty * (T_{md} + T_{wb})$$

where *Nb_read_miss_on_clean* is the number of read misses for which the memory copy is clean when the request is received and *Nb_read_mis_on_dirty* is the number of read misses for which the memory copy is stale.

In these experiments, Radiosity has a very large data set size and its instruction miss rate is higher than all other benchmarks, which in turn results in up to 18% instruction stall time on an 8-processor system. This high instruction miss rate may be partially explained by the nondeterministic behaviour of Radiosity. There may also be some trashing between data and instruction accesses. MP3D has the worst behavior over all. In MP3D the memory controller utilization of board 0 reaches 94.5% on 8 processors. Due to the large number of messages generated by MP3D the memory controller becomes a hot spot and the read and write stall times increase with each additional processor. When we implement page level memory interleaving in RPM, we expect to reduce the utilizations of the memory controllers. Even though it is not as efficient as it can be, the current RPM configuration is also an actual implementation of a shared memory multiprocessor and MP3D nicely illustrates the hot spot problem on such machines.

Figure 8 (d) shows the FutureBus+ utilization. This utilization is computed from the event

counts using a similar procedure as for the memory controllers. The FutureBus+ is very much underutilized. Therefore, we can safely assume that we have an infinite bandwidth interconnection for all practical purposes. In order to emulate limited bandwidth systems we can artificially increase the size of each packet sent to the FutureBus+.

5.2. Varying Data Set Sizes

We have also examined the effects of increased data set sizes for Barnes-Hut, Water-nsquared, Ocean, Radix, and FFT on a 4-processor system. For Barnes-Hut we have used 4K, 8K, and 16K particles. For Water-nsquared the number of molecules were changed from 512 up to 1331. We ran Radix with 256K, 1M, 2M, 4M, 8M and 12M integers with a radix of 1024. For FFT 64K, 256K, and 1M points are used. We simulated Ocean with 130x130, 258x258 and 514x514 grid sizes. The memory used by these benchmarks are 8, 16, and 32 MB for Barnes-Hut, 550KB, 775KB, 1MB, and 1.4MB for Water-nsquared, 2, 8, 16, 32, 64, and 96 MB for Radix, 3, 12, and 48 MB for FFT, and 3.7, 14, and 56 MB for Ocean.

When increasing the data set sizes we were limited by three factors: 1) Execution time 2) Memory size 3) Completion of the program execution. Among the selected benchmarks Radix, Ocean, and FFT are memory-bound. On a 4-processor RPM we have 128 MB memory and for these three benchmarks we could not run bigger problems because of our memory limitations. Water-nsquared and Barnes-Hut are compute-bound programs and we are currently limited by their execution time. In the current RPM configuration we are limited to about two hours of execution time because of the 32-bit resolution of the event counters in count memory. In the near future we plan to interrupt the IO processor periodically every two hours to freeze the whole system, upload all the counters to the host Sun SPARC Workstation, and resume the program. We expect this mechanism to add very little distortion to the collected performance statistics (it takes a couple of pclocks to stop all processors) while increasing the tolerance to crashes. We had enough memory and event-counter bits to run Barnes-Hut with 32K particles, but unfortunately the system crashed after running for one hour. We are currently fixing this problem.

Figure 9 and Figure 10 show the total and the normalized execution times of the five benchmarks on four processors. The run for Ocean 514 took the longest time, about one hour and 20 minutes on RPM. The busy time is the total execution time including instruction executions when all data read/write accesses hit in the First Level Cache. The Null Busy time is the sum of all the times when the processor can not execute a useful instruction because of an pipeline interlock or a unused branch delay slot. The write stall, read stall, swap stall and instruction stall times are the times during which a processor is blocked pending the completion of a write, a read, a swap and an instruction fetch in the memory system. The spin time is the total synchronization time excluding the swap stall time.

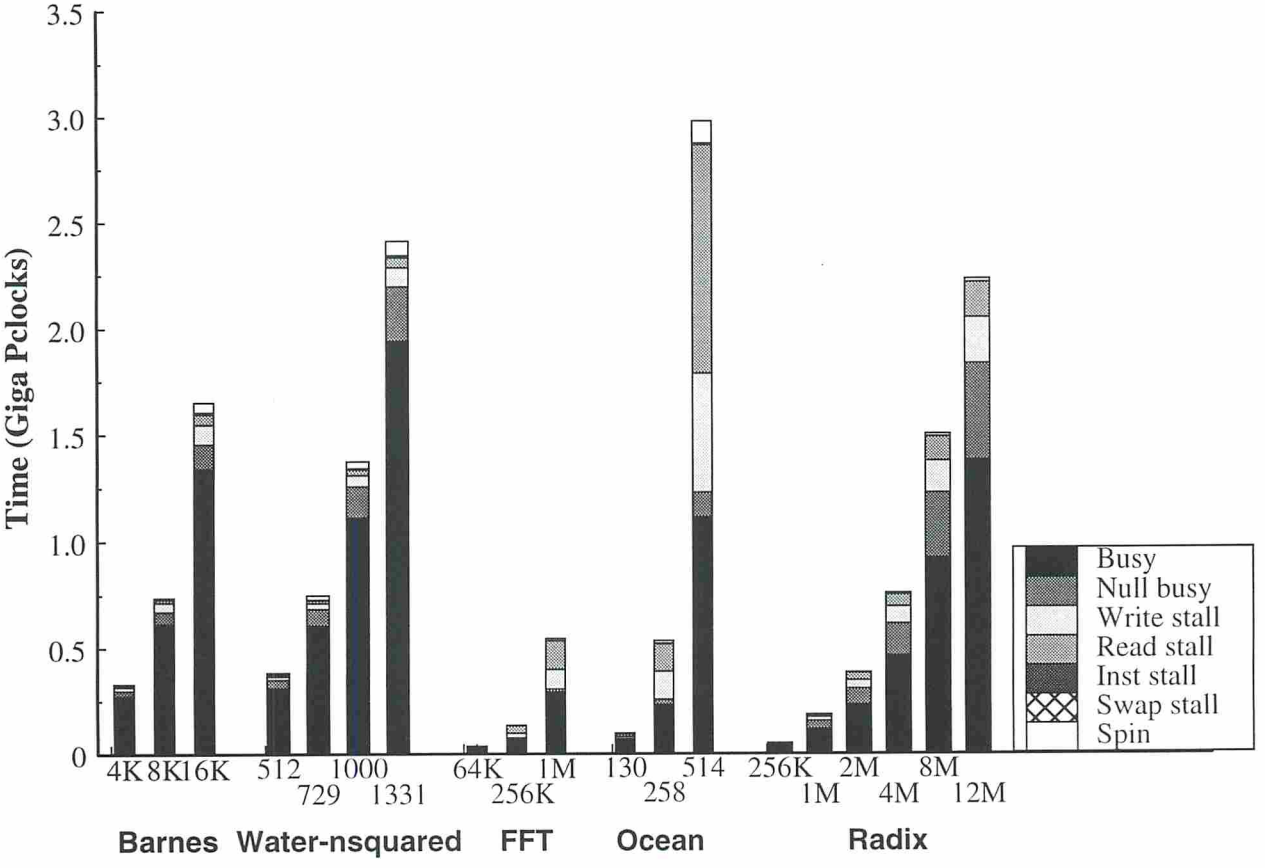

Figure 9 Total Execution Times of the Five Benchmarks on a 4-processor System

Barnes-Hut and Water-nsquared have very high processor utilizations and their memory behavior is not affected by the increased data set sizes, because their working set sizes are much

smaller than 1 MB and therefore they fit in our second-level cache easily. Pipeline overheads dominate.

The memory behavior of Radix is also unaffected by data set size increases. This result can be explained mainly by the communication-to-computation ratio of Radix. This ratio is constant as long as the number of processors is fixed. One interesting point about Radix is the large amount of time it spends in Null Busy cycles. A large number of branch delay slots are not utilized. In this case, processor overheads dominate memory overheads.

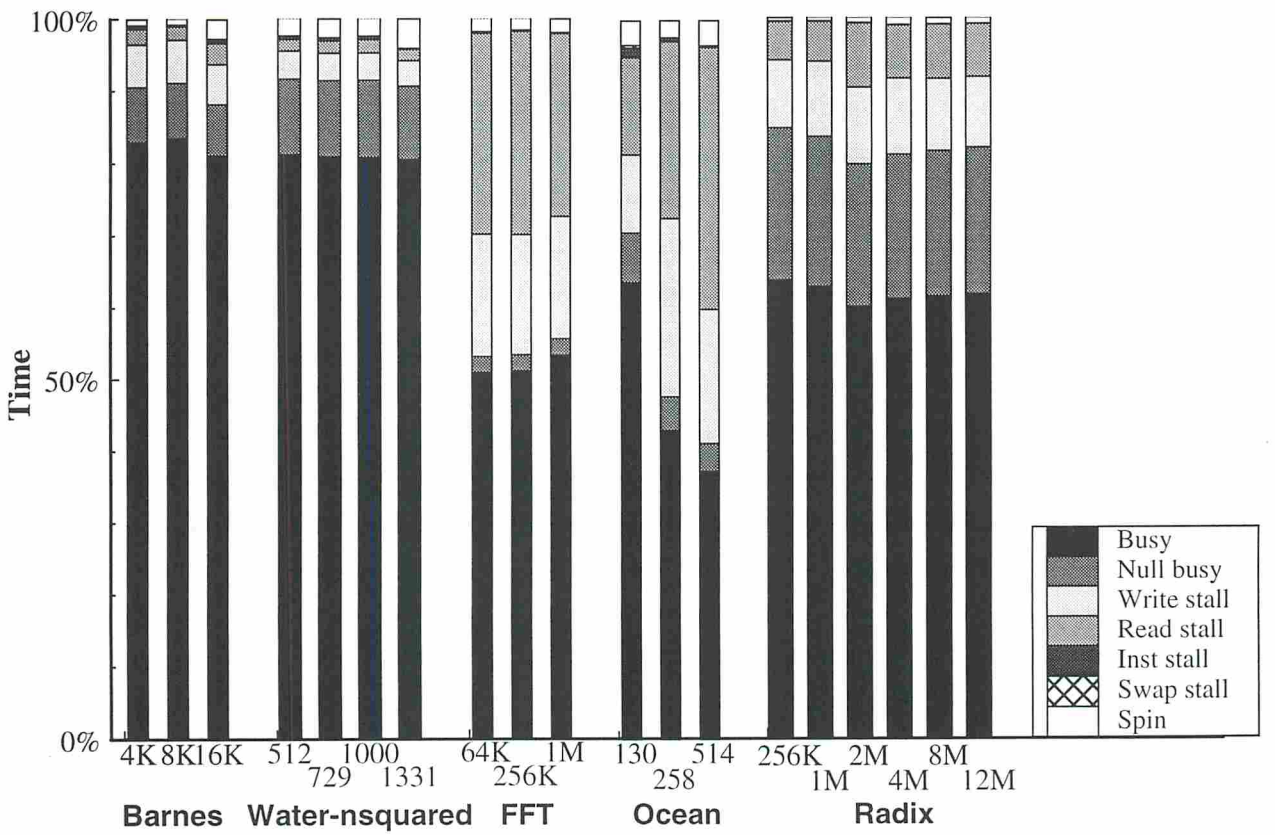

Figure 10 Normalized Execution Time of the Five Benchmarks on a 4-processor System

FFT is another benchmark whose memory behavior is unaffected by increasing the data set sizes. In FFT the most important data set size is one row of the matrix [29], which fits our second level cache. On top of that the communication-to-computation ratio decreases logarithmically with increasing data set sizes. We have run FFT with three data set sizes: 64K points (3 MB),

256K points (12 MB), and 1M points (48 MB). The changes in communication-to-computation ratio from 64K points to 256K points and from 64K points to 1M points are 0.9152 and 0.8437, respectively. We see the slight effect of the reduced communication-to-computation rate in terms of increasing busy times for larger data sets.

Ocean is the only benchmark which shows a dramatic difference in its memory behavior as the data set size is increased. Ocean has a large working set size which does not fit into our 1 MB second-level cache for the 514 x 514 grid size. (It is possible that the working set size for the 258 x 258 grid does not fit in our second-level cache as well.) As the data set size increases the miss rates in the second-level cache increases from 0.4% to 1.3% and then to 2.3%. As reported in [29], the local data traffic increases drastically with the data set size for 4-processor configurations and a 16 bytes block size. Since the memory in RPM is allocated contiguously starting from board 0, some data which is supposed to be local to a processor may be allocated in another board's memory, which increases the service time of the read/write misses to local data as well as the data traffic.

5.3. Simulation Experiments

Table 2 shows some comparisons between RPM emulations and RPMsim simulations. All the experiments reported in Table 2 have been conducted for four processors clocked at 5MHz. The simulations were performed on a SUN SparcStation-10 with 128 Mbytes of memory (a 50 MIPS machine). The time values correspond to the parallel section of the benchmark only. The average emulation speedup is close to 170 and it should more than double when eight processors are used instead of four. Also, this speedup should double when we reach the 10MHz clock for which the boards were designed. Applications with high busy times (Barnes-Hut, Volrend, Raytrace, Water) achieve large emulation speedups.

These speedups are given here as an indication. In fact, more efficient simulation approaches could cut the simulation time. Also, the simulation could be parallelized or run on faster machines. Still, even taking these factors into consideration, the speedups are quite impres-

sive. The simulation time depends heavily on the level of simulation details. The emulation speedups would be much higher if the emulation was compared to cycle-by-cycle simulation.

The simulation accuracy ranges from surprisingly good (Barnes-Hut, Cholesky, FFT, LU, Raytrace, Water) to moderate (Radix, Volrend) and bad (Radiosity). In the emulation, Radix spends 15 to 20% of its time in null busy cycles, which are unaccounted for in the simulation. Volrend makes heavy use of mathematical library functions, which are not simulated. Hence the simulated execution times are too optimistic. Finally, Radiosity seems to have followed different convergence paths to the solution in the simulation and in the emulation. Clearly, we will have to include pipeline effects in the simulation and simulate more of the mathematical library functions if we want to improve the accuracy of the simulator

Benchmark	Parameters	Emulation time(sec.)	Simulation time (sec.)	Emulation speedup	Parallel run time (10^6 clk.)	Simulated parallel run time (10^6 clk.)	Simulation accuracy (%)
barnes	4k	533	94483	177.3	333.1	343.1	+3.0
cholesky	bcsstk14	37	6546	176.9	23.5	24.5	+4.3
fft	-m16 -n4096	50	6444	128.9	31.3	31.7	+1.3
lu	-n128	10.4	2163	207.9	6.5	6.9	+6.2
radiosity	test	848	105083	123.9	529.9	324.5	-38.8
radix	262144 int.	74	11253	152.1	46.4	40.8	-12.1
raytrace	teapot	252	49973	198.3	157.3	166.2	+5.7
volrend	scaleddown4	70	11588	165.5	43.8	35.2	-19.6
water-nsq	343	303	56498	186.5	189.4	192.3	+1.5

Table 2 Comparison of Emulation with Simulation

6. CONCLUSION AND FUTURE WORK

Multiprocessor emulation is an alternative to prototyping and software simulation. In this paper we have related our initial experience with emulation technology based on field-programmable gate arrays for the performance evaluation of multiprocessor systems. RPM is a configurable hardware platform or substrate on top of which various multiprocessor architectures can be imple-

mented in hardware. Because the controllers are made of FPGAs, architecture parameters as well as performance measurement hardware can be easily changed. The emulation and performance methodologies are based on multi-cycle pclock emulation for flexibility, time scaling, and event-count memory. We have described the methodologies and illustrated them with our first CC-NUMA emulation on RPM. Currently we have a complete software environment to run parallel scientific programs based on the ANL macros.

By using Saavedra's micro-benchmark approach, we are able to determine the cache parameters and the latencies of various accesses. We were surprised of how close these experimental values were to the estimated values based on hardware cycle counts, since some hardware components do not always have fixed delays. The micro-benchmarks have been an extremely valuable tool to gain confidence in the validity of the design and the accuracy of the latency estimates.

In RPM every single event is emulated, including instruction fetches, and pipeline effects. We have seen in our emulations that these often neglected effects may in some cases dominate other performance overheads. Clearly there are some obvious improvements to the CC-NUMA hardware. One is to add a memory mapping table in the first level cache to relocate data and instructions to the physical memory in a flexible way. Another easy improvement is to overcome the limitations on the emulation time due to the limited 32-bit size of the event counters, by uploading the counters periodically. Obviously, we also need to improve the reliability of the RPM simulator so that we can rely on it for debugging the performance measurement on RPM.

We are currently developing a COMA emulator [14], so that we can compare it with the CC-NUMA. To port system software on RPM requires that we add hardware support for virtual memory and that we boost the I/O capabilities of RPM. We have also to develop the time scaling methodology for I/O processing if we want to make interesting system-level measurements on RPM.

Acknowledgments. Funding for this work has been provided by the National Science Foundation under Grant MIP-9223812. Alain Gefflaut is funded by INRIA/IRISA (France). Besides the authors, several individuals have contributed to the project. In particular, we want to thank Per Stenström from Lund University (Sweden) and Massoud Pedram from EE-Systems, U.S.C. Luiz Barroso, Jacqueline Chame, Sasan Iman, and Krishnan Ramamurthy participated in the design of the RPM emulator. Through their University Program several companies helped reduce the cost of the hardware and software needed for the project. These companies are Advanced Micro Devices, Synopsis, Viewlogic, Axil Workstations and Xilinx. Finally, John Granacki from ISI offered the services of EZFAB, which is part of the ARPA-sponsored Systems Assembly Project.

7. REFERENCES

[1] Agarwal, A., Bianchini, R., Chaiken, D., Johnson, K.L., Kranz, D., Kubiatowicz, J., Lim, B.-H., Mackenzie, K., Yeung, D., "The MIT Alewife Machine: Architecture and Performance," *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[2] 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.

[3] Barroso, L.A., and Dubois, M.,"Performance Evaluation of the Slotted-Ring Multiprocessor," *IEEE Transactions on Computers*, Vol. 44, No. 7, pp.878-890, July 1995.

[4] Barroso, L.A., Iman, S., Jeong, J., Öner, K., Ramamurthy, K., and Dubois, M., "RPM: A Rapid Prototyping Engine for Multiprocessor Systems," *IEEE Computer*, pp. 26-34, February 1995.

[5] Blumrich, M.A., Li, K., Alpert, R., Dubnicki, C., Felten E., and Sandberg, J.,"Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.

[6] 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.

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

[8] Censier, L., and Feautrier, P.,"A New Solution to Coherence Problems in Multicache Systems," *IEEE Transactions on Computers*, Vol. 27, No. 12, pp.112-118, December 1978.

[9] Convex, "Examplar Architecture". Convex Press, 1993.

[10] 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.

[11] 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.

[12] Dubois, M., Skeppstedt, J., and Stenström, P., "Essential Misses and Data Traffic in Coherence Protocols," *Journal of Parallel and Distributed Computing*, October 1995.

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

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

[15] Kuck, D., et al., "The Cedar System and an Initial Performance Study," *Proceedings of the*

20nd International Symposium on Computer Architecture, May 1993.

[16] Kuskin, J., et al. "The Stanford FLASH Multiprocessor," *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.

[17] 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.

[18] 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.

[19] Muller, H.L., Stallard, P.W., Warren, D., and Raina, S.,"Parallel Evaluation of a Parallel Architecture by means of Calibrated Emulation," *Proceedings of the 8th Int. Parallel Processing Symposium, pp. 260-267*, April 1994.

[20] Noakes, M.D., Wallach, D.A., and Dally, W.J., "The J-Machine Multicomputer: An Architectural Evaluation," *Proceedings of the 20nd International Symposium on Computer Architecture*, May 1993.

[21] Nowatzyk, A., Aybay, G., Browne, M., Kelly, E., Parkin, M., Radke, B., and Vishin, S.,"The S3.mp Scalable Shared-Memory Multiprocessor," *Proceedings of the 1995 International Conference on Parallel Processing*, pp. I.1-10, August 1995.

[22] Öner, K., Barroso, L.A., Iman, S., Jeong, J., Ramamurthy, K., and Dubois, M., "The Design of RPM: An FPGA-based Multiprocessor Emulator," *Proceedings of the 3rd ACM International Symposium on Field-Programmable Gate Arrays*, June 1995.

[23] Pong, F., Stenström, P., and Dubois, M.,"An Integrated Methodology for the Verification of Directory-based Cache Protocols," *Proceedings of the 1995 International Conference on Parallel Processing*, pp. I.158-166, August 1994

[24] Reinhardt, S., Hill, M.D., Larus, J.R., Lebeck, A.R., Lewis, J.C., and Wood, D.A., "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.

[25] Saavedra, R.H., Gaines, R.S., and Carlton, M.J., "Micro Benchmark Analysis of the KSR1", *Proceedings of Supercomputing'93,* Portland, November 1993.

[26] Schwetman, "CSIM: A C-based, Process-oriented simulation Language," *Proceedings of 1986 Winter Simulation Conference*, pp. 387-396, 1986.

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

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

[29] Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., and Gupta, A.,"The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.