

# Collecting Memory Address Traces from an Ericsson Cell Phone and Estimating Cache Performance

## Technical Report CESR-TR-01-1

**Ashwini Sidhaye, Paul Steinmetz, Eric Rotenberg**

Center for Embedded Systems Research (CESR)  
Department of Electrical and Computer Engineering  
North Carolina State University  
{aasidhay,pmsteinm,ericro}@ece.ncsu.edu  
[www.tinker.ncsu.edu/ericro](http://www.tinker.ncsu.edu/ericro), [www.cesr.ncsu.edu](http://www.cesr.ncsu.edu)

**David Barrow, Domenico Arpaia**

Ericsson, Inc.

## 1. Introduction

This project involves characterizing cache performance of a mobile phone. Specifically, we will recommend cache configurations based on simulation, identify performance bottlenecks, and isolate behavior unique to real-time embedded systems that can be exploited for higher performance with efficient implementations. A key aspect of the project is building tracing infrastructure. Memory address traces are collected using data acquisition hardware/software connected to a phone emulator board.

In the first year, summarized in this technical report, we focused on building the tracing infrastructure for the system. We also performed validation of the traces and experimented with them using a cache simulator.

This report is organized as follows.

- Emulator board

The emulator board discussion is further divided into three parts: the processor, which describes the ARM7TDMI microprocessor used by the emulator board; the memory interface, which enables monitoring addresses; and the procedure for setting up the cell phone for making calls.

- Gathering traces using logic analyzer

This section outlines the procedure used to gather small sample traces using a Tektronics TLA 704 logic analyzer.

- Gathering traces using PC data acquisition

This section is divided into two parts, software and hardware for large-scale data acquisition. The software section includes descriptions of two software tools – LabView and VC++ – that are used to program the PC data acquisition card. The

hardware section describes the hardware setup for acquiring the address traces from the emulator board.

- Trace validation

PC data acquisition is powerful, but we need to ensure the traces are a reliable representation of the emulator board's memory trace. This section talks about the ways in which the traces were validated. Validation includes (1) comparison with known-correct logic analyzer samples, (2) visual inspection, and (3) confirmation of expected cache behavior (miss rate trends vs. size, associativity, etc.) using cache simulations. Finally, we discuss remaining problems with accuracy of the traces.

- Benchmark creation and description

This section describes arbitrary sequences of dialing and other phone manipulation used to create "benchmarks". Benchmark creation is work-in-progress.

- Cache simulation setup

Two methods for carrying out the cache simulations are described. The first approach dumps traces to large disk files, and then cache simulation is performed by reading traces from disk. The second approach embeds the cache simulation code inside the trace gathering software, consuming traces on-the-fly.

- Experimental results

This section presents preliminary results that we obtained by running cache simulations using the traces collected from the emulator board.

## **2. Emulator board**

The emulator board is a real cell phone mounted on a printed circuit board. This enables various internal signals to be probed for execution trace gathering.

### **2.1 Processor**

The cell phone uses an ARM7TDMI microprocessor. This processor can execute the 32-bit ARM and 16-bit THUMB instruction sets. Information on a generic ARM7TDMI processor can be found by going to <http://www.arm.com/sitearchitek/armwww.ns4/html/documentation?OpenDocument> and selecting the ARM7TDMI under "Technical Reference Manuals". This document describes the instruction set architecture.

For the purposes of this project, the most relevant information is that instructions can be either 32 bits or 16 bits, depending on the mode the processor is in. The processor can change between modes at any time, but spends the majority of its time running in the 16-bit THUMB mode. The processor does not have a cache

(this project investigates the performance implications of using a cache), although some ARM processors do have them.

## 2.2 Memory interface

The primary concern of this project is the memory interface. The cell phone uses an 8-bit data bus and a 21-bit address bus. Refer to the *Emulator Board Schematic* in Appendix C for the pin names. The address uses pins 4 through 19 on jumper J1101 and pins 7 through 11 on jumper J1102.

There are four additional pins we are concerned with: Output (Read) Enable (pin 3 on J1101), Write Enable (pin 3 on J1102), Chip Select 1 (pin 4 on J1102), and Chip Select 2 (pin 6 on J1102). Chip Select is required to enable the memory device, and acts like a global enable to the device. Output Enable is required to enable the output data bus of the memory device (Chip Select must also be active). Finally, Write Enable is required to write data to the memory devices, and again Chip Select must also be active. Writes usually occur on the trailing edge of a Write Enable pulse. Note that Output Enable and Write Enable should never occur together.

We have been using the following signals to capture addresses using the logic analyzer and PC DAQ card, respectively.

- *Logic analyzer*: Trigger on Chip Select 1 active and the rising edge of Output Enable (the data should still be valid until a short time after Output Enable goes inactive).
- *PC DAQ card*: The Output Enable is used as a clock for sampling. The address bus is sampled at every Output Enable edge, as was done with the logic analyzer. Neither Chip Select is examined, however. When this method was used, segments of the trace matched the small trace gathered from the logic analyzer.

We have since learned that there are really two memory devices, independently enabled by Chip Select 1 and Chip Select 2. One memory device is SRAM for data memory and the other is Flash for instruction memory. Given this information, we suspect that the traces gathered using the PC DAQ card contain (1) all instruction addresses and (2) load addresses. We come to this conclusion because the Chip Selects are not examined, so we do not distinguish between instructions (Flash) and data (SRAM). Also, we do not examine Write Enable, meaning we do not include memory stores. Only memory loads are captured by virtue of examining Output Enable. Section 5.5 summarizes future work regarding separating instruction and data addresses and capturing both loads and stores.

## 3. Gathering traces using logic analyzer

We used a Tektronics TLA 704 logic analyzer. It runs Windows 95 and is very easy to program and use. The only drawback is that it collects a maximum of 16 thousand samples, which is not enough for realistically measuring cache performance. The following procedure explains how to hook up the emulator board and collect an address trace from it (see last paragraph in Section 2.2 for caveat –

we suspect the method below gathers only instruction addresses by triggering only on Chip Select 1 and Output Enable). Documentation for the TLA 704 can be found in 301 EGRC.

1. Connect +5 Volts to the RED connector.
2. Connect -4.5 Volts to the YELLOW connector. This is used to power the display. Changing the voltage up or down slightly will affect the contrast.
3. Connect Ground to the BLACK connector. Make sure the ground for the two power supplies are connected here.
4. Connect the logic probes to the board. Use Appendix C to see which pins connect to which probe. Connect the grounds from the probes to pin 20 on J1101 and J1102. You will have to use the clips to connect the two grounds to J1101. Make sure the large cables connect to A0, A1, and A2 on the TLA 704.
5. Make sure the current limiters on the power supplies are set to at least 2 amps.
6. Turn on the 704 Logic Analyzer and wait for the program to start. The user name and password are "tek".
7. Go to File->Load System and select "ericsson".
8. Turn on the power supplies. If the cell phone display is not visible, disconnect and reconnect the -4.5 Volt supply.
9. Maximize the window titled "Setup".
10. Click on the green "run" button. When it stops, the trace will have been collected.
11. Select the "Listing" window. Go to File->Export Data to save the trace as a text file.
12. Using Windows, copy the text file to a floppy.
13. Copy the file from the floppy to a Sun station or PC. The file is now ready to be used with the cache simulator.

Since this analyzer can only collect a very short trace, we began looking at alternative methods of data collection and decided to use the PCI-DIO-32HS DAQ device. Note that, while the logic analyzer is not used for gathering full traces, it is invaluable for validating larger traces captured with the more powerful but difficult-to-verify PC tools (refer to Section 5.1).

## **4. Gathering traces using PC data acquisition**

### **4.1 Software**

#### **4.1.1 LabView software for gathering memory traces**

We decided to first use LabView 5.0, a graphical programming tool. Using LabView as a starting point is helpful because it comes with simple example programs, which were used immediately to confirm that the card could sample voltages. In fact, we

tested the card by tying one or more pins to +5V and confirmed that 1's were read by the pre-packaged programs.

Below, we document the steps taken to setup the software (so that future students may re-trace our steps).

1. We downloaded NI DAQ 6.9 from the National Instruments (NI) website and installed the driver and the solution wizard on the PC.
2. The 301 lab has a LabView 5.0 CD which we used to install LabView on the PC. We chose the English component of LabView Version 5.0 and used the channel wizard installer to install all 4 disks.
3. The Measurement and Automation Explorer is used to configure the PC card.

For more information regarding details for configuring the PC card using Measurement and Automation Explorer, channel wizard installer, and LabView refer Appendix D.

Configuring the PC card using the Measurement and Automation Explorer. The Measurement and Automation Explorer searches for new devices installed on the PC and helps configure them.

1. Go to Start -> Programs -> National Instruments -> Measurement & Automation.
2. There will be a tree structure on the left side of the window. Click on the Devices and Interfaces.
3. The DAQ board PCI-DIO-32HS will be listed under it.
4. On the right hand side you will see the DAQ Device Basics.
5. Click on Run the DAQ Test panels options. Some information will appear beneath it.
6. Right click on the device PCI-DIO-32HS. Click on the Test Panel option. Check the error codes to find out whether any error has occurred.

We stopped using LabView because C++ is more flexible and it becomes possible to embed the cache simulator inside the trace gathering code.

#### **4.1.2 C++ software for gathering memory traces**

NI-DAQ libraries provide a set of C functions for interacting with the NI-DAQ card. Thus, we can write and compile a C/C++ program, linked with NI DAQ library routines, to collect traces. Using C/C++ also enables embedding other functionality, such as cache simulation code, in the same program. Note that different sample programs are available in the NI-DAQ reference manual, which can aid writing the software.

The programming environment available on the PC is Visual C++. To get familiar with the VC++ programming environment, follow the steps outlined below.

1. Click on Start -> Program Files -> Microsoft Visual Studio 6.0 -> Microsoft Visual C++ 6.0.
2. To create a workspace go to File -> New -> Projects -> Win 32 Console Application.
3. Give a name to the project. You will be asked, "What kind of Console Application do you want to create?". Choose the option 'An empty project' and then click on Finish.
4. A dialog box will appear on the screen. Click OK.
5. A workspace will be created for you. Right click on the Source Files folder and then click on 'Add files to folder...'. This way, you can insert the source files for the project in the folder.
6. Right click on the Header Files folder and then click on 'Add files to folder...'. Insert the header files required for your code.
7. If a workspace has already been created, then go to File -> Open and then open the .dsw file of the required workspace.
8. Go to Project -> Settings. Click on the 'Link' tab. Add the following two paths to the Object/Library modules textbox:
  - "c:/Program Files/National Instruments/NI-DAQ/Lib/nidaq32.lib"
  - "c:/Program Files/National Instruments/NI-DAQ/Lib/nidex32.lib"
  - Note: If you are using any other libraries, then they also need to be specified.
9. Go to Project -> Settings. Click on the 'C/C++' tab. Select 'Preprocessor' in the Category drop-box. Specify the path for the additional header files, if any, in the 'Additional Include Directories' textbox.
10. Click OK when finished.
11. Go to Build -> Execute to compile and run the .exe file for the project. A console window appears and you can input values in it.

For a list of manuals used and descriptions of NI-DAQ library routines refer to Appendix D.

The flowchart in Figure 1 describes the operation of the trace collection program at a high-level, highlighting calls to key NI-DAQ routines. Cache simulation code is optionally embedded within the program (more on this below). The full C++ program is included in Appendix A.

Before arriving at the correct program in Figure 1, we tried many variations. We made four key discoveries in our search for a well-behaved trace gathering program.

1. The best operating mode for the NI-DAQ card in our application is *external pattern generation mode*. "External" refers to using an external clock to direct when to sample, and the cell phone's Output Enable is used in this case (just like we did with the logic analyzer). There are better modes than pattern generation mode, for example, burst mode can achieve much higher transfer rates. However, burst mode requires flow-control between the PC and the emulator board. The emulator board does not have provisions for stalling it via PC control. Pattern generation mode does not use flow control and is therefore our best choice.

2. Pattern generation mode works best in conjunction with the “asynchronous” function call DIG\_Block\_In (other modes prefer different input functions). DIG\_Block\_In initiates filling a buffer with addresses sampled by the NI-DAQ card. The next item below has more information about the implications of “asynchronous” functions such as DIG\_Block\_In.
3. There is a required “polling loop” that continuously checks the status of the buffer until it is determined that the buffer is full of samples. The polling loop is shown as a test block in the second page of the flowchart. The polling loop and asynchronous function DIG\_Block\_In are related. An “asynchronous” function is one, which initiates something on the NI-DAQ card but immediately returns control back to the calling program. Therefore, the polling loop is needed to stall the program until the NI-DAQ card is actually finished with the requested operation, in this case filling the buffer with address samples.
4. The Visual C++ programming environment, like many compilers, restricts the size of buffers that can be allocated both statically and dynamically. Thus, even though VC++ software allows larger buffers than the logic analyzer, we are still somewhat constrained (10 million addresses, for example). So, we used a large but restricted buffer. Then, to generate a 100 million address trace, we use an outer loop that repeats the process many times. This outer loop is shown with a dashed arrow in the flowchart of Figure 1. A problem with this approach is there may be large gaps in the address trace between iterations, because the C++ program does not provide any real-time guarantees. While ending the previous iteration and beginning a new iteration, the cell phone continues to operate and we don't know how many samples escape the C++ program during this transition period. The problem is exacerbated when we insert cache simulation code between iterations.

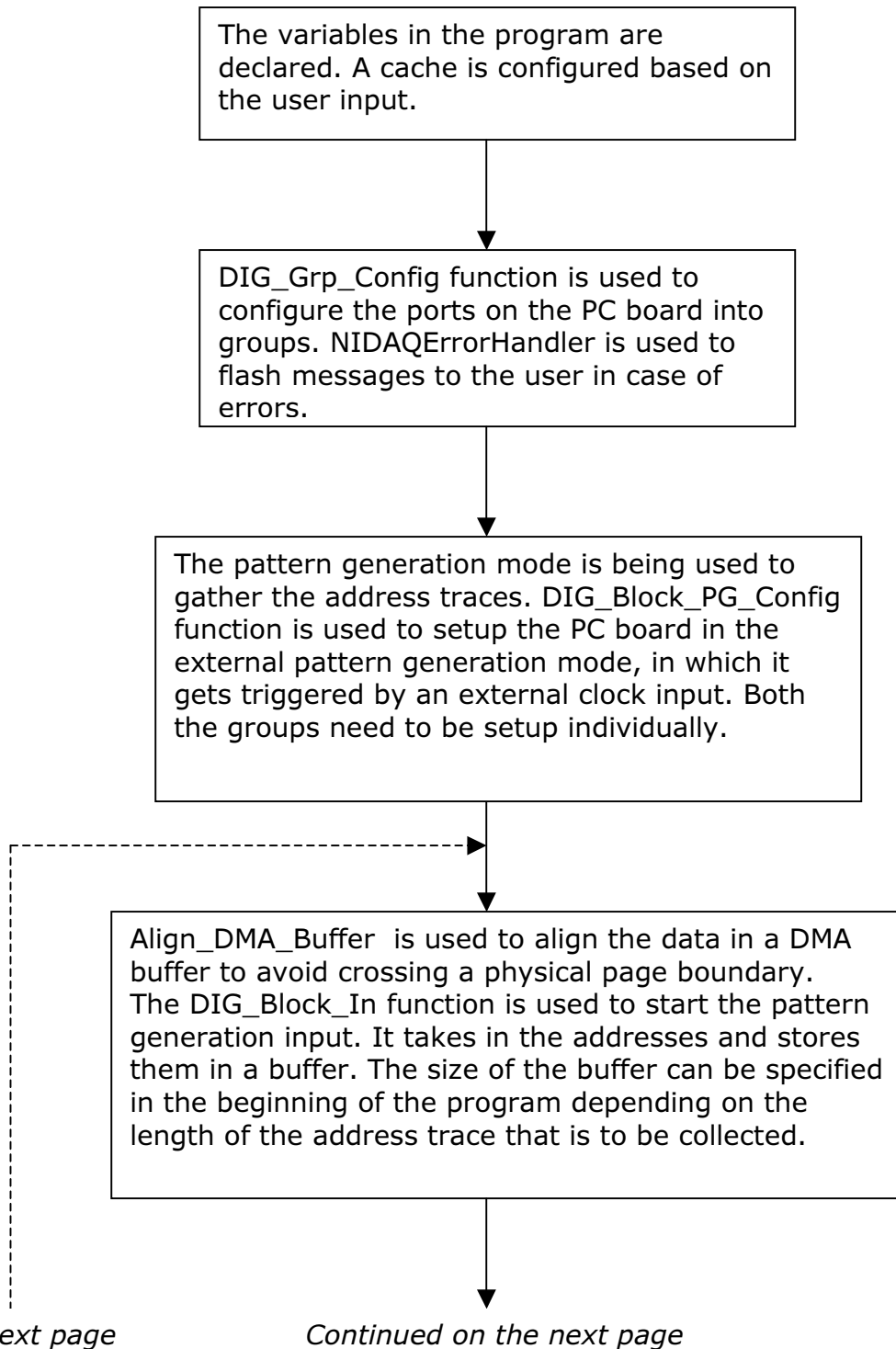
Calls to the cache may optionally be embedded in the trace-gathering program. Furthermore, there are two methods for embedding calls to the cache. The first method is to gather a buffer-worth of addresses, then go through the buffer sequentially in a subsequent loop to pass the addresses to the cache simulator. This is shown in the flowchart with a box labeled “*The addresses are read from the buffer and input to the cache simulator.*” Note, this inserts more delay between trace-gathering iterations and exacerbates the problem highlighted in point #4 above.

A second approach interleaves cache simulation with the polling loop (shown in the flowchart with a curvy arrow). Instead of waiting for the entire buffer to be full before sending addresses to the cache, we can send whatever addresses have been sampled to the cache. Embedding cache simulation code within the polling loop reduces between-iteration overhead, potentially reducing the size of gaps in the address trace. We have not yet implemented this approach but conjecture that it is feasible. In parallel, we are also investigating optimizing the cache simulation code to make it more efficient and reduce its performance overhead<sup>1</sup>.

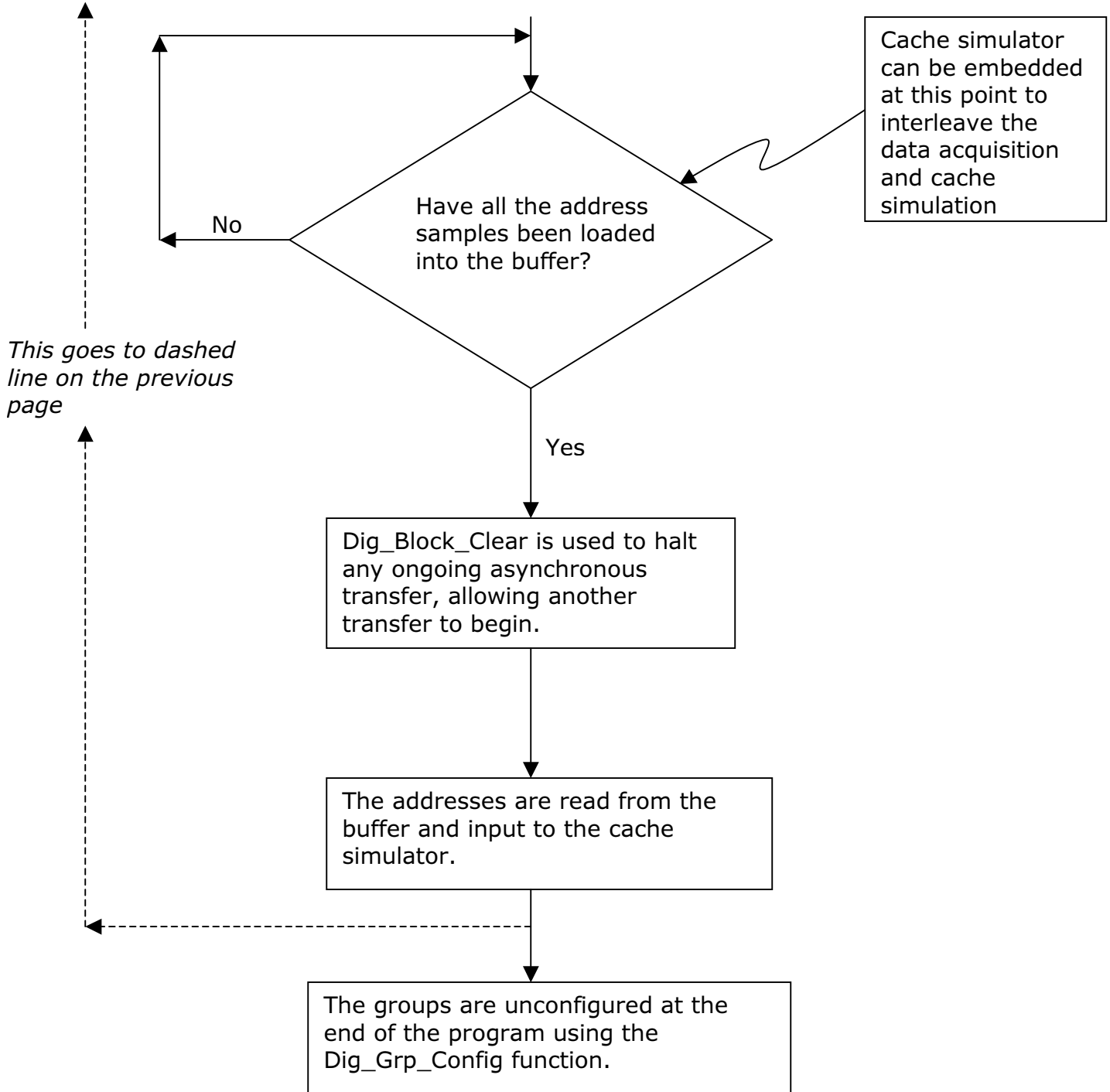
---

<sup>1</sup> In the late 80s and early 90s, Mark Hill at the University of Wisconsin – Madison developed an efficient method for simulating many caches simultaneously, termed the *dinero* cache simulator; we may leverage that work.

**Figure 1. Flowchart of the address trace collection program, with cache simulator embedded (optional).**



*Continued from the previous page*



## 4.2 Hardware system design

The first task is physically installing the PC card in the computer. The PC cover is removed via a push-button located on the left-hand side of the front panel, near the bottom. Pushing this button pops open the cover. There is an electronic manual on the PC hard drive that gives directions for installing PCI cards.

For the manuals refer to Appendix D.

After physically installing the PC card, we installed the necessary software (drivers) to make it functional. Download 'Nidaq69.exe' from the National Instruments website, run the .exe file, and follow the instructions given by the installer.

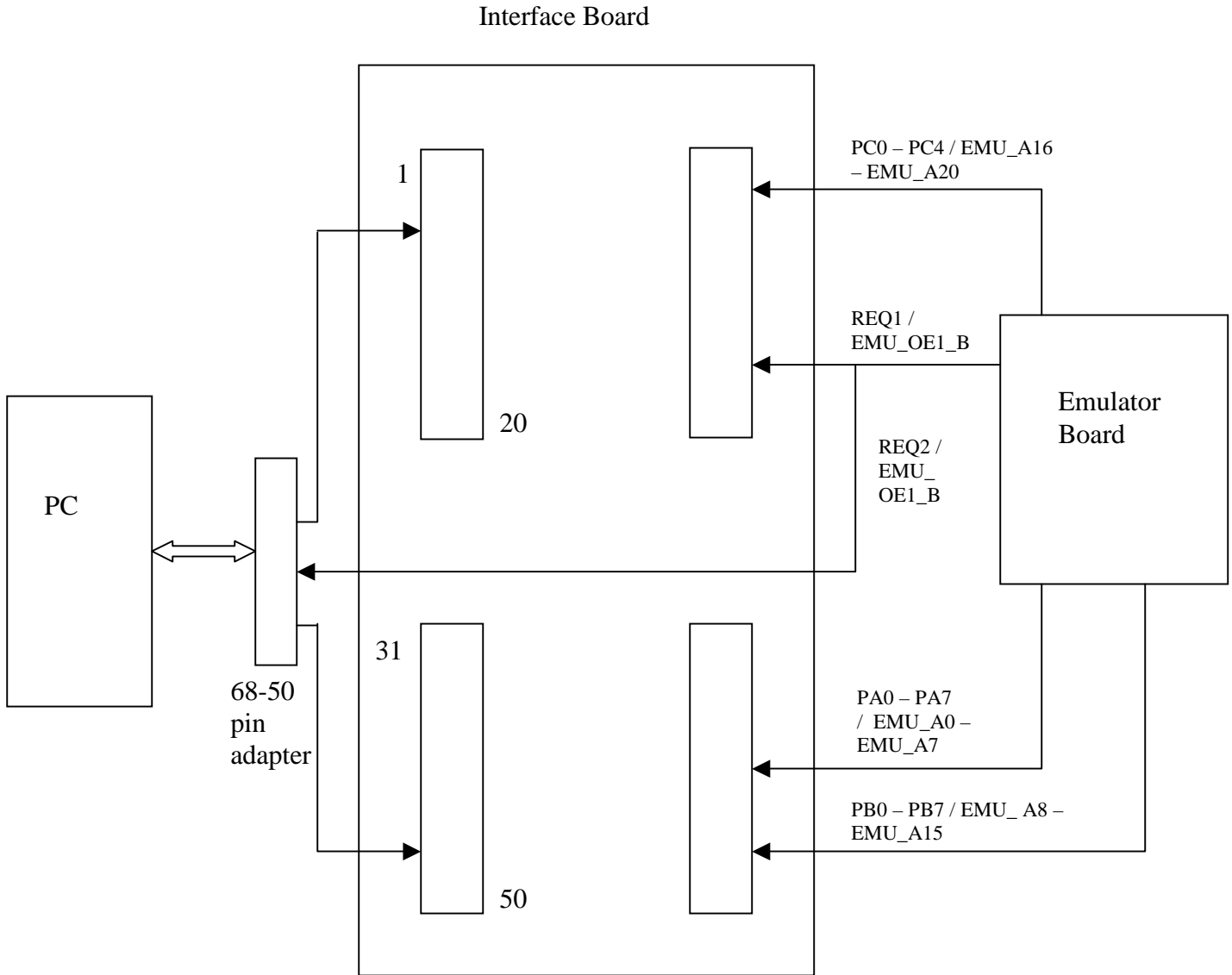
Figure 2 shows a block diagram of the trace collection hardware. It consists of three major parts.

1. The PC side, which includes the NI-DAQ card and the 68-to-50-pin adapter.
2. A custom-designed interface board for interfacing the PC to the emulator board.
3. A description of signals coming from the memory interface on the emulator board and how they connect to the custom interface board.

The following table defines the signals in the block diagram of Figure 2.

Signal name on emulator board	Signal name on interface board	Signal name on PC card	Signal type	Description
EMU_A0 – EMU_A7 <i>(bits 0 to 7 of address)</i>	PA0 – PA7	DIOA<0..7>	Data	Port A bi-directional data lines. Port A is port number 0. DIOA7 is the MSB; DIOA0 is the LSB.
EMU_A8 – EMU_A15 <i>(bits 8 to 15 of address)</i>	PB0 – PB7	DIOB<0..7>	Data	Port B bi-directional data lines. Port B is port number 1. DIOB7 is the MSB; DIOB0 is the LSB.
EMU_A16 – EMU_A20 <i>(bits 16 to 20 of address)</i>	PC0 – PC7	DIOC<0..7>	Data	Port C bi-directional data lines. Port C is port number 2. DIOC7 is the MSB; DIOC0 is the LSB. (Note: we use only 5 lines of Port C, so only those are shown.)
EMU_OE1_B <i>(Output Enable)</i>	REQ1, REQ2	REQ<1..2>	Control	Group 1 and group 2 request lines (a group is configured to consist of up to 16 bits, e.g., two 8-bit ports). In pattern gen. mode, REQ lines strobe data into or out of the PC card.

**Figure 2. Overall block diagram of trace collection hardware.**



### **4.2.1 PC side**

The PC has a National Instruments (NI) data acquisition (DAQ) card installed in it (hence the term NI-DAQ card). The specific name of the card is PCI-DIO-32HS, and it belongs to a class of cards commonly called 6533 devices. The PCI-DIO-32HS has a 68-pin I/O connector. This can't be connected directly to a flat ribbon cable (FRC) or commonly-used FRC connectors. An optional 68-to-50-pin adapter can be used to overcome this problem. We purchased this adapter because it enables us to better interface with the other circuitry. The female side of the adapter has 68 pins and connects directly to the PCI-DIO-32HS, and the male side provides 50 pins at the output. This is connected to a FRC. The 50-pin side has no +5V, CPULL, or DPULL pins and has fewer ground pins than the 68-pin side. But overall, we do not lose any important signals in the conversion from 68 to 50 pins.

Figure 3 shows the pin assignments of the PCI-DIO-32HS 68-pin I/O connector. Figure 4 shows the pin assignments of the 68-to-50-pin adapter.

**Figure 3. Pin assignments of the PCI-DIO-32HS 68-pin I/O connector.**

DIOD7	34	68	GND
GND	33	67	DIOD6
DIOD4	32	66	DIOD5
DIOD3	31	65	GND
GND	30	64	DIOD2
DIOD0	29	63	DIOD1
DIOC7	28	62	GND
GND	27	61	DIOC6
DIOC4	26	60	DIOC5
DIOC3	25	59	GND
GND	24	58	DIOC2
DIOC0	23	57	DIOC1
DIOB7	22	56	RGND
DIOB6	21	55	GND
GND	20	54	DIOB5
RGND	19	53	DIOB4
GND	18	52	DIOB3
DIOB1	17	51	DIOB2
DIOB0	16	50	GND
DIOA7	15	49	GND
GND	14	48	DIOA6
DIOA4	13	47	DIOA5
DIOA3	12	46	GND
GND	11	45	DIOA2
DIOA0	10	44	DIOA1
REQ2	9	43	RGND
ACK2 (STARTTRIG2)	8	42	GND
STOPTRIG2	7	41	GND
PCLK2	6	40	CPULL
PCLK1	5	39	GND
STOPTRIG1	4	38	DPULL
ACK1 (STARTTRIG1)	3	37	GND
REQ1	2	36	GND
+5 V	1	35	RGND

**Figure 4. Pin assignments of the 68-to-50-pin adapter.**

DIOD1	1	2	DIOD4
DIOD3	3	4	DIOD0
DIOD6	5	6	DIOD7
DIOD2	7	8	DIOD5
DIOC5	9	10	DIOC7
DIOC3	11	12	DIOC1
DIOC2	13	14	DIOC0
DIOC6	15	16	DIOC4
GND	17	18	ACK2
GND	19	20	STOPTRIG2 (IN2)
GND	21	22	PCLK2 (OUT2)
GND	23	24	REQ2
GND	25	26	GND
ACK1	27	28	GND
STOPTRIG1 (IN1)	29	30	GND
PCLK1 (OUT1)	31	32	GND
REQ1	33	34	GND
DIOA4	35	36	DIOA6
DIOA0	37	38	DIOA2
DIOA1	39	40	DIOA3
DIOA7	41	42	DIOA5
DIOB5	43	44	DIOB2
DIOB7	45	46	DIOB6
DIOB0	47	48	DIOB3
DIOB4	49	50	DIOB7

### **4.2.2 Custom-designed interface board**

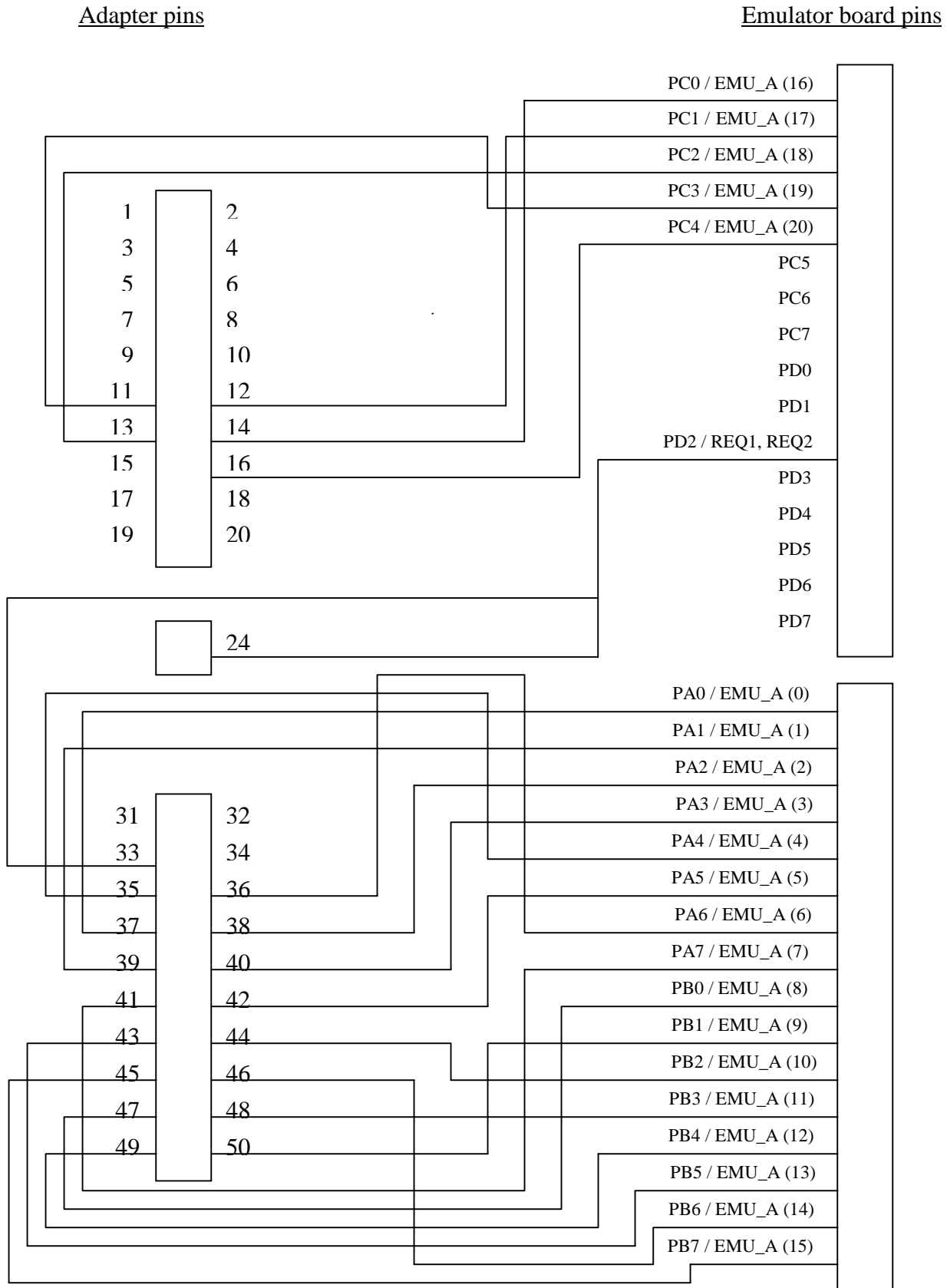
In order to connect the address lines on the emulator board to the computer's 68-to-50-pin adapter, we built an interface board which has the 68-to-50-pin adapter connected to one side and the address lines that come from the emulator board lined up on the other side. Figure 5 shows the interconnect layout of the custom-designed interface board.

In order to connect the 68-to-50-pin adapter to the interface board, we designed a simple cable. We used a conventional flat ribbon cable and crimped a 50-pin header on one end and two 20-pin headers on the other end. The 50-pin header connects directly to the 68-to-50-pin adapter. The two 20-pin headers connect directly to the left-hand side of the interface board as shown in Figure 5.

Finally, either a flat ribbon cable or individual wires connect signals from the emulator board directly to the right-hand side of the interface board as shown in Figure 5.

Note: Pin 24 is shown separately as it is not a part of the header pins. It has been connected separately to the emulator board.

**Figure 5. Interconnect layout of the custom-designed interface board.**



### **4.2.3 Emulator board memory interface**

The third component of the trace gathering hardware is the emulator board itself, more specifically, its memory interface. Memory interface signals and how they are used were described to some extent in Section 2.2 ("Memory Interface"). However, here we describe (1) connecting the Output Enable to REQ strobes for sampling addresses, (2) connecting address lines to three 8-bit hardware ports of the NI-DAQ card, and (3) grouping hardware ports into "groups" via software routines.

The NI-DAQ card is used in pattern generation mode. In this mode, sampling is timed by request pulses carried on the REQ pin. The NI-DAQ card can generate request pulses internally or we may instead provide external pulses. We do the latter since the emulator board indicates when an address is on the address lines via the Output Enable control signal. To be complete, we should also monitor the Write Enable control signal, otherwise we only capture load addresses and not store addresses; see Section 2.2 for a discussion of this and other issues.

When dealing with the NI-DAQ card, signals are managed both at the hardware-level and software-level. At the hardware-level, signals are managed in 8-bit quantities called "ports". The PCI-DIO-32HS (as the name suggests) has a bandwidth of 32 bits. The 32 bits are divided into 4 8-bit ports called Port A (numerically, port 0), Port B (numerically, port 1), Port C (numerically, port 2), and Port D (numerically, port 3). Recall, the table at the beginning of Section 4.2 described how we use Port A, Port B, and part of Port C. We need to monitor 21 address bits. The first 16 address bits go to Ports A and B and the remaining 5 address bits go to Port C.

At the software-level, signals are managed in "groups". A group is configured by software routines. We can configure a group to contain one or two 8-bit hardware ports. For example, we configured Group 1 to contain Ports A and B (16 bits total) and Group 2 to contain Port C (8 bits total). This allows us to sample up to 24 bits (currently we only need 21 bits, but in the future we may also want to sample the cell phone's data bus, or at least the 2 chip selects).

Groups are mentioned because they are relevant for understanding the REQ1 and REQ2 strobe signals. REQ1 is the strobe for Group 1 and REQ2 is the strobe for Group 2. We connected the Output Enable to both REQ1 and REQ2, since in reality we only want one strobe for all the address bits.

Confused yet? The program in Appendix A gets everything right.

## **5. Trace validation**

### **5.1 Examining traces visually and comparisons with the logic analyzer**

In this section, we describe a simple initial method for ensuring traces collected by the PC card are reliable. Traces are suspect for several reasons. First, it took

awhile to come up with a good program due to the complexity of understanding how to program and use the NI-DAQ card. The logic analyzer trace gave us an idea of what valid output should look like. Without this, we would have been working in the dark. We compared the first 16K samples against the samples obtained by the logic analyzer. Eventually, we were able to match closely and observed repeating sequences of incrementing addresses – these are common memory access patterns (incrementing the program counter or scanning through an array, redirecting control flow due to branches, loops, etc.).

The second reason the traces are suspect is that our trace gathering hardware and software system does not provide hard real-time guarantees, unlike the logic analyzer. The NI-DAQ card is real-time to some extent but the PC and its software, being a general-purpose system, provides no such guarantees (for example, the PCI bus may be disrupted by other unrelated processes in the system). We have noticed small glitches in the trace, for example, two adjacent addresses being the same. Remaining problems with traces are described in Section 5.4.

The third reason traces are suspect was already described in Section 4.1.2. We pointed out how limited buffering in the C++ program, and the resulting need to iterate, may introduce large gaps in the trace. The cell phone continues to produce addresses while the trace gathering program transitions to a new iteration. Currently, we do not have a means to measure how large the gaps are (or if they indeed exist) and this is work-in-progress.

In summary, we successfully matched large segments of the trace obtained from the PC with the trace obtained from the logic analyzer. Although only 16K samples are available from the logic analyzer, we expect repetition throughout the large trace and verified this visually.

Note that it is impractical to view one large trace file using editors, so the trace was split across many small disk files to enable visual inspection.

## **5.2 Cache simulation tests and miss rate analysis**

Another common sanity-check technique is looking for expected and reasonable behavior. In our case, this means performing cache simulation and measuring miss rates.

First, the absolute miss rate, while not totally verifiable by simulation alone, should not be unusually high or low. Second, cache size, set-associativity, and cache line size all affect the miss rate in known ways. For example, increasing cache size and set-associativity generally reduces miss rate, with diminishing returns after some point. Cache simulation results are presented in Section 8.

## **5.3 Validating the cache simulator**

We also performed validation of the cache simulator itself. We did this by comparing two different versions of the simulator (this is called N-version

programming in the fault-tolerant software community). The two different but functionally-equivalent simulators were developed by Ashwini and Eric. We verified that both simulators produce the same miss rate for a given trace.

## 5.4 Remaining trace problems

The problems that we continue to work on include:

- (Also see Section 4.1.2.) Alleged large gaps: limited capacity of the C++ buffer forces us to iterate and, in turn, transition time between iterations potentially introduces gaps in the trace. This is exacerbated by cache simulation overhead between iterations, which may be partially alleviated by overlapping sampling code and cache simulation code.
- (Also see Section 5.1.) Small glitches such as repeating consecutive addresses. We can bind the error introduced by this particular glitch by combining identical consecutive addresses (we call this a “filter” mechanism).

## 5.5 Other issues

As mentioned in Section 2.2, currently we are only using Output Enable as a strobe for gathering traces but we need to look at other signals as well. There are three types of address – instruction addresses, load addresses, and store addresses. Output Enable (signaling a memory read) captures only instruction and load addresses. We need to use Write Enable to capture store addresses. Also, we need to look at the two Chip Selects to distinguish between instruction addresses and load/store addresses.

We have not verified that all of the above is completely accurate but it is a starting point for future work (for example, we need to understand which chip select is for the Flash memory and which is for the SRAM memory; we need to verify that Flash contains instructions and SRAM processes loads and stores; etc.).

## 6. Benchmark creation

Currently, we gather traces while the phone is “idle” (not processing a call). This is the simplest benchmark and an important one.

Future work includes creating other benchmarks based on typical cell phone usage. The simplest example is dialing into the cell phone and then hanging up (this simulates hanging up immediately on a telemarketer!). Below is an example list of arbitrary benchmarks we plan to gather memory traces for. Benchmarks will then be selected or discarded based on memory behavior that we want represented in the overall benchmark suite.

1. Dial into the cell phone from the lab phone. Say 3 sentences separated by 3-second intervals.
2. Dial out to the lab phone then say 3 sentences separated by 3-second intervals from the lab phone.

3. Dial into the cell phone from another cell phone. Say 3 sentences separated by 3-second intervals.
4. Dial out to another cell phone then say 3 sentences separated by 3-second intervals from the other cell phone.
5. Dial into the cell phone from the lab phone. Say 3 sentences separated by 3-second intervals. Interrupt with another cell phone. Switch over to the other call and say one sentence. (This assumes call-waiting support.)
6. Press a few keys on the keypad.

## **7. Cache simulation setup**

### **7.1 Stand-alone cache simulator**

The cache simulator is derived from the cache simulators developed in ECE 521. It has been modified so it will work with a different input file format. When the cache is run stand-alone (not embedded in the trace gathering program), the trace that is stored on disk is given as input to the cache simulator. Also, the simulator prompts the user to enter cache configuration parameters – cache size, block size, and set-associativity. When the simulator finishes, it prints out statistics, including the miss rate and number of accesses. It also performs limited, automated trace-anomaly detection (such as detecting identical consecutive addresses) and produces relevant statistics. The miss rate is used to plot the graphs.

### **7.2 Merging tracing code and cache code**

The cache simulator can be called from within the tracing code by embedding the cache simulator source code within the trace-gathering program.

The primary advantage of this approach is avoiding dumping large trace files to disk. From a practical standpoint, large trace files are difficult to manage. For example, it takes a long time to transfer the traces among different computer systems, which we sometimes want to do based on the location of various simulators. Breaking the large trace file into many smaller files is sometimes helpful but does not fix all problems.

From a technical standpoint, not writing traces to disk means cache simulations can run for trillions of addresses because there is no space constraint. Also, consuming traces on-the-fly often runs faster than reading large disk files, despite having to re-create the trace every time simulation is performed.

There are several disadvantages to merging cache simulation with trace gathering. First, extra delay is inserted between trace-sampling iterations, possibly increasing the duration of gaps in the address trace. Second, repeatability becomes an issue. That is, precisely re-creating a particular benchmark is impossible. Instead, we must rely on averaging and statistical cache simulation. Furthermore, if we want to compare the performance of different cache configurations, all caches must be simulated simultaneously to ensure the same address stream is applied to all

caches. The results in Section 8 were generated by simulating all cache configurations at the same time. Finally, embedding the cache simulator means performing cache simulation requires having possession of the emulator board and tracing infrastructure, making it difficult to share address traces among researchers/developers.

## **8. Experimental results**

In this section, we present cache miss rates for many cache configurations. Cache size is varied from 4KB to 2MB. Direct mapped, 2-way set-associative, and 4-way set-associative caches are simulated. Two different cache block sizes are simulated, 16 bytes and 32 bytes.

The cache simulator is embedded within the trace gathering program. To ensure the same trace is applied to each cache in the same graph, all caches are simulated during the same run of the program.

Each trace contains 100 million addresses. The trace is gathered while the cell phone is in idle mode (not processing a call).

### **8.1 Trace variance**

First, we wanted to determine trace variation. That is, we performed one simulation consisting of 100 million addresses and repeated for a second simulation. Results for the two runs, trace #1 and trace #2, are shown in Figures 6 and 7, respectively. Notice the two graphs show similar performance for all cache configurations. This means there is little variation for the idle-mode benchmark and this benchmark is fairly repeatable. Since Figure 7 is essentially a repeat of Figure 6, the next section focuses on the graph in Figure 6.

### **8.2 Miss rate as a function of size and set-associativity**

The graph in Figure 6 shows miss rate as a function of cache size and set-associativity, for a block size of 16 bytes. The first observation is that increasing cache size reduces the miss rate. We normally expect a smooth monotonically decreasing curve with diminishing returns. We observe this trend in Figure 6, but we also observe two distinct regions where this is the case. For the direct mapped cache, there is a smooth monotonically decreasing region until 128KB, after which there is a steep decline. Apparently, when the direct mapped cache size exceeds 128KB, one or more program constructs (instructions and/or data) begins to fit in the cache and the miss rate drops more steeply. Further analysis of per-address misses would reveal which particular instructions and addresses are the cause of this behavior (this is future work).

From Figure 6, the second observation is that increasing set-associativity from direct mapped to 2-way set-associative decreases miss rate substantially, from around 15% to 7% for a 64KB cache. Miss rate further decreases when set-

associativity is increased from 2-way to 4-way, but the absolute reduction is less than before, matching our expectations of diminishing returns.

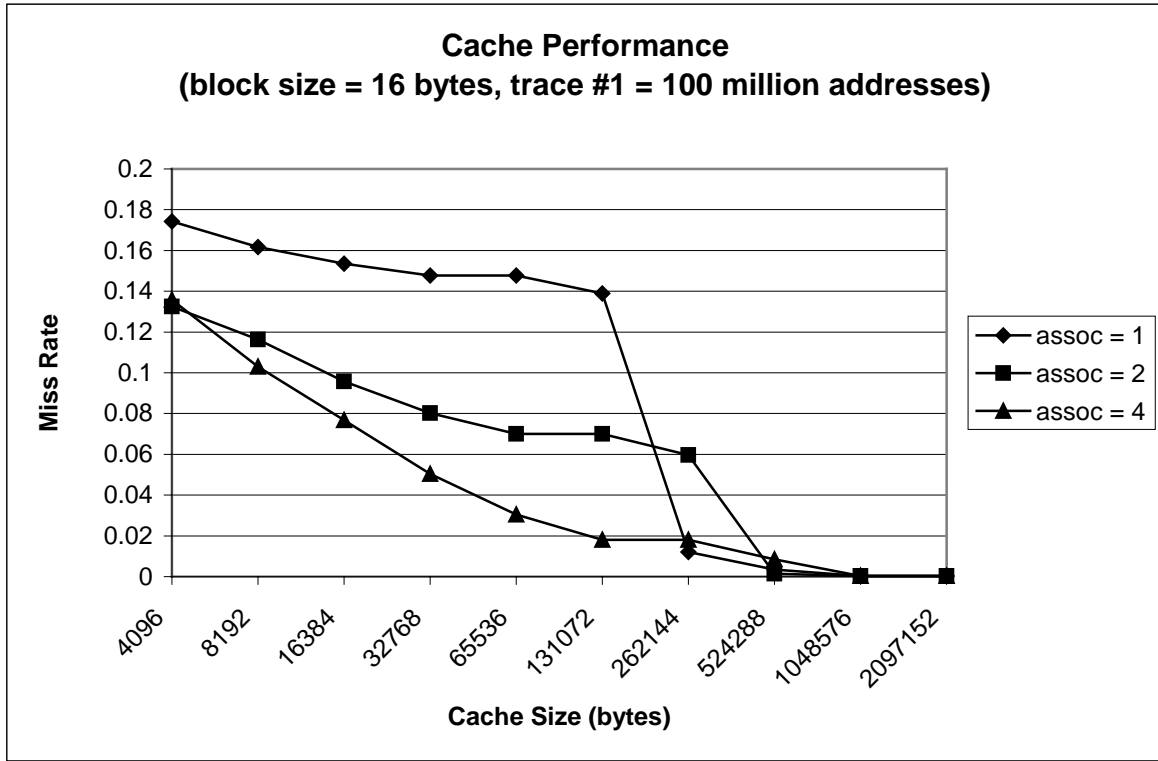
Finally, we perform several other "sanity checks" of the data in Figure 6. First, the miss rates are not exceptionally high or low. However, we are somewhat surprised a cell phone produces measurable miss rates for very large caches. In future work, we will attempt to understand why this is the case. Simulation of separate instruction and data caches will be more revealing than the unified cache simulations performed here.

The second "sanity check" has to do with the cell phone's memory capacity. Because the memory system is limited to 21 address bits, a 2MB cache should fit all accesses without any conflicts. So, there should be no capacity or conflict misses for a 2MB direct mapped or set-associative cache, only compulsory misses, and we expect the miss rate to be very small. Figure 6 confirms this expectation. We also checked the trace for any addresses that exceed the address range of the cell phone, and did not observe any.

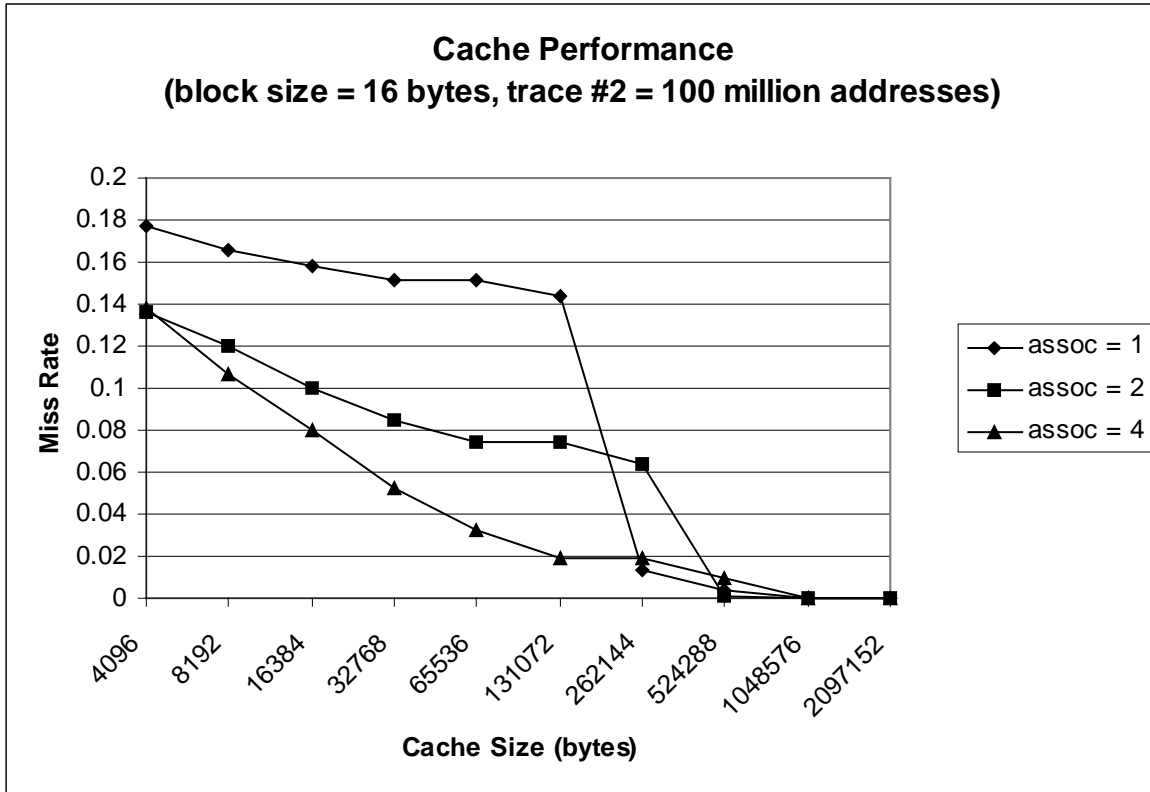
There is one interesting anomaly in the data of Figure 6: the miss rate of a 256KB direct mapped cache is less than the miss rate of a 256KB 2-way (and even 4-way) set-associative cache. After that point, the miss rates tend to converge towards 0. Without per-address cache profiling, we are unable to explain this anomaly. However, associativity has been known to produce anomalous behavior, although we are surprised the anomaly is so large and we will certainly use address profiling to uncover the reason.

The graph in Figure 8 repeats the 100 million address trace experiment for a block size of 32 bytes. Note, Figure 8 uses a different trace than Figure 6 (since the experiment was re-run). However, Section 8.1 leads us to believe that trace variance tends to be small.

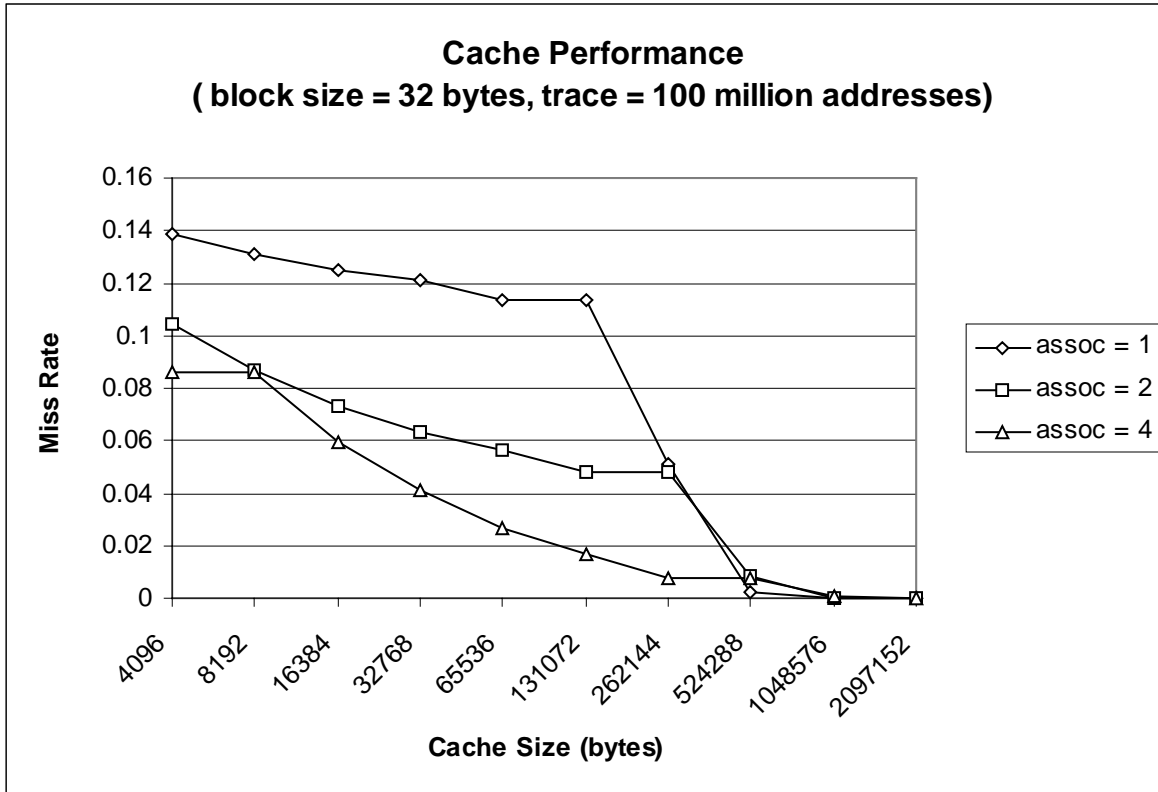
Figure 8 (32 byte block size) shows similar trends as Figure 6 (16 byte block size): larger cache size and higher set-associativity decrease miss rate, both show smooth monotonically decreasing miss rates with diminishing returns, etc. Also, as we expect, larger block size tends to reduce miss rate due to better spatial locality. This aspect is investigated in more detail in the next section. Finally, the anomaly seen in the 16B block size case (described above) is much less pronounced in the 32B block size case.



**Figure 6. First cache simulation using 100 million addresses. Block size is 16 bytes and cache size and set-associativity are varied.**



**Figure 7. Measuring trace variance: this is a repeat of the first experiment in Fig. 6 and shows similar miss rates.**

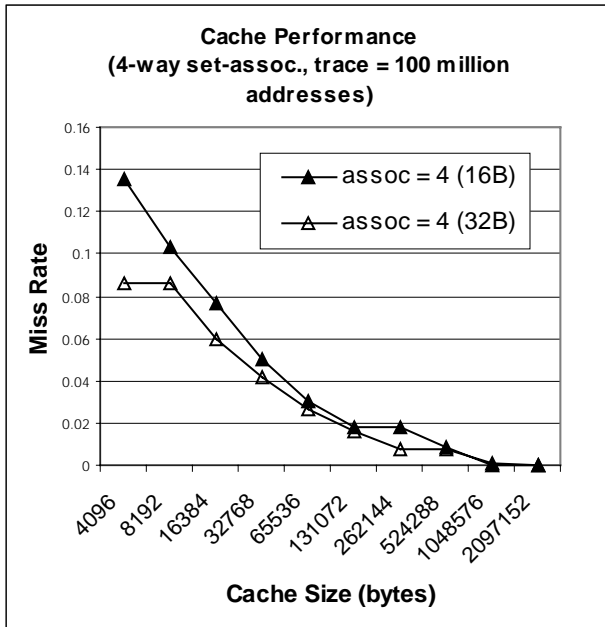
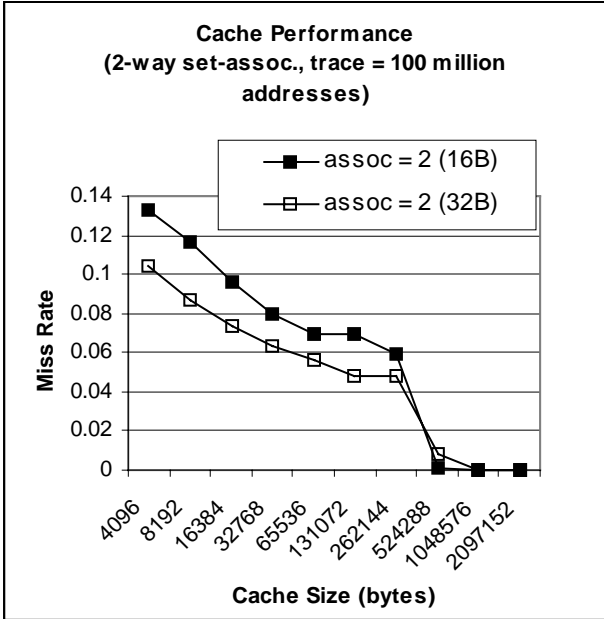
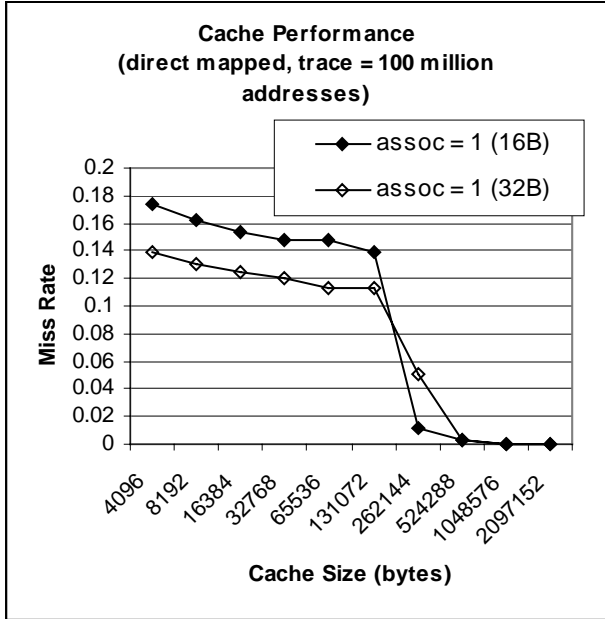


**Figure 8. Third cache simulation using 100 million addresses. Block size is 32 bytes and cache size and set-associativity are varied.**

### 8.3 Effect of block size on miss rate

Figure 9 shows the effect of block size — 16 bytes (16B) versus 32 bytes (32B) — for each of direct mapped, 2-way set-associative, and 4-way set-associative caches. Except for a few anomalous cases, 32 byte block size performs better than 16 byte block size, presumably due to better spatial locality.

For the direct mapped cache, before the steep drop in miss rate at 256KB, the absolute miss rate delta for the two block sizes is fairly constant at about 4% (for example, 14% versus 18% for a 4KB direct mapped cache). The delta tends to be slightly smaller for 2-way and 4-way set-associative caches. Overall, we conclude block size has a large impact on miss rate. However, set-associativity and (at certain points) cache size have a larger impact.



**Figure 9. Effect of block size (16B vs. 32B) is shown for each of direct mapped, 2-way set-associative, and 4-way set-associative caches.**

## 9. Summary, Conclusions, and Future work

This project involved characterizing cache performance of a mobile phone. The tracing infrastructure system was developed, which involved gathering memory address traces from the emulator board using data acquisition methods. Two different data acquisition methods were used, namely:

- 1) Data acquisition using the Logic Analyzer (LA).
- 2) Data acquisition using the PC card.

The traces acquired were then validated by visual inspection, comparison of the traces obtained using the two methods, and analyzing them to check whether they displayed the expected cache behavior.

Various signals like Output Enable (OE), Write Enable (WE), Flash\_CS, SRAM\_CS, and EEROM\_CS coming from the emulator board were used to capture addresses using the PC data acquisition card. We found out that OE and WE separated the reads and writes to the memory devices respectively, and the chip selects enabled us to distinguish between the instructions and data.

The PC card data acquisition method was used to gather the traces for the experiments, as the traces acquired using the LA were not long enough. The pattern generation mode in conjunction with the asynchronous function DIG\_Block\_In was chosen to collect the traces, as the PC card did not use flow control. DIG\_Block\_In, being asynchronous, made it necessary to use a polling loop, which would stall the program until the NI-DAQ card finished filling the buffer with the required number of address samples.

The software used to program the PC card is Visual C++, which does not provide any real-time guarantees. The Visual C++ programming environment, like many compilers, also restricts the size of buffers that can be allocated, both statically and dynamically. So, in order to collect large traces, we used multiple iterations. Due to the non-real-time environment, this poses a problem, as large gaps might occur in the address trace between the iterations.

We suggested two methods for embedding the cache code within the trace-gathering code. The first collects a buffer-worth of addresses and then passes them to the cache simulator, and the second interleaves the cache simulation with the polling loop so that the addresses are simulated by the cache as soon as they are sampled. In this technical report, we used the first method.

On the other hand, merging cache simulation with trace gathering has several disadvantages. First, the extra delay inserted between trace-sampling iterations may increase the duration of gaps in the address trace. Interleaving simulation in the polling loop reduces the problem. Second, precise re-creation of a particular benchmark is not possible. As we had to compare the performance of different cache configurations, all caches had to be simulated simultaneously to ensure the same address stream was applied to all caches. Still, little variation was observed for the idle-mode benchmark and it was found that the benchmark is fairly repeatable.

After carrying out the cache simulations, miss rate analysis is performed, with the following key results.

- ❑ The absolute miss rate is not unusually high or low. Increasing cache size and set-associativity reduces miss rate, with diminishing returns after some point. The emulator board has 21 address bits, so logically a 2MB cache should fit all accesses without any conflicts, and our experimental results confirmed this. All of these are successful sanity checks for the experiments.
- ❑ Block size has a large impact on miss rate. However, set-associativity and (at certain points) cache size have a larger impact.
- ❑ For the direct mapped cache, a smooth monotonically decreasing region is observed until 128KB, after which there is a steep decline.
- ❑ Increasing set-associativity from direct mapped to 2-way set-associative decreases miss rate substantially, from around 15% to 7% for a 64KB cache. Miss rate further decreases when set-associativity is increased from 2-way to 4-way, but the absolute reduction is less than before, matching our expectations of diminishing returns.
- ❑ One unexplained anomaly was observed in the data of Figure 6: the miss rate of a 256KB direct mapped cache is less than the miss rate of a 256KB 2-way (and even 4-way) set-associative cache. After that point, the miss rates converged towards zero.

Future extensions to this work include the following.

1. We will use complete signals for properly distinguishing loads, stores, and instruction fetches, and then simulate unified, instruction, and data caches. << NOTE: This has been completed and documented in the addendum. >>
2. We will perform per-address miss rate profiling to gain better insight into results, and also explain the few anomalies. The per-address profiling should also be tied back to the source code or some other high-level representation of the program.
3. We will simulate other benchmarks.

## 10. Addendum: Tracing support for separate unified, instruction and data caches

### 10.1 Chip select table

OE	WE	Flash_CS	SRAM_CS	EEROM_CS	Comments
0	0	---	---	---	Never occurs
0	1	0	0	0	Never occurs
0	1	0	0	1	Never occurs
0	1	0	1	0	Never occurs
0	1	0	1	1	Reads from Flash: Instruction address.
0	1	1	0	0	Never occurs
0	1	1	0	1	Reads from SRAM: Load.
0	1	1	1	0	Reads from EEROM: don't care (rare).
0	1	1	1	1	Rare: unexplained.
1	0	0	---	---	Never occurs
1	0	1	---	0	Never occurs
1	0	1	0	1	Writes to SRAM: Store.
1	0	1	1	1	Never occurs
1	1	---	---	---	Never occurs

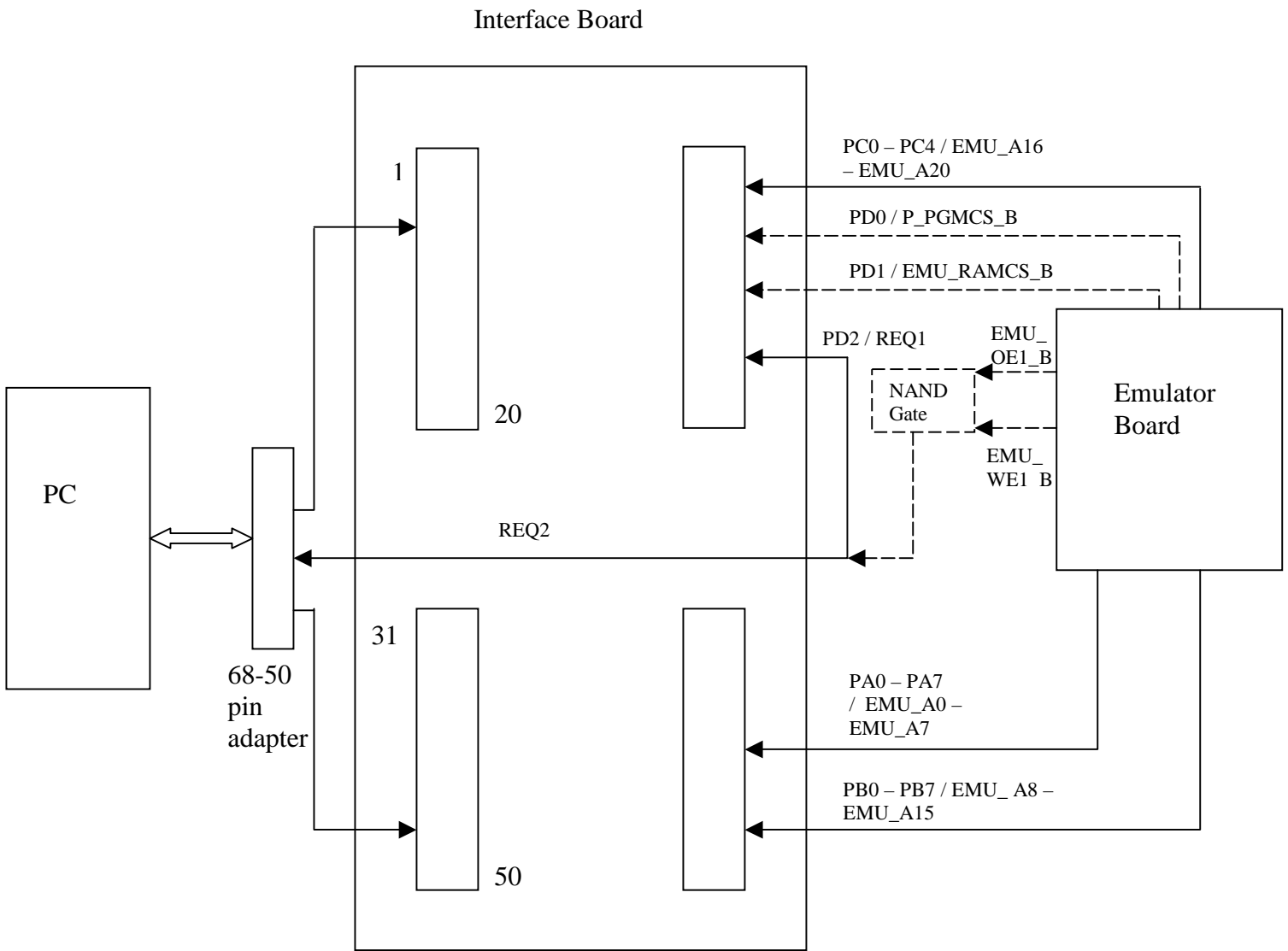
During the initial phase of the work, we used only the OE signal to capture instruction and load addresses. The table presents a complete picture of the memory system with the three chip selects, namely, Flash\_CS, SRAM\_CS, and EEROM\_CS, and the Output Enable (OE) and Write Enable (WE) signals. The Flash\_CS and SRAM\_CS help us distinguish between accesses to instructions and data, respectively. OE, which signals a memory read, signals the instruction and load addresses. The WE signals store addresses. OE and WE are never simultaneously active or inactive. We observe that the Flash memory, SRAM, and EEROM can be read but only the SRAM can be written into. We verified that this table occurs in practice using the logic analyzer.

## 10.2 Modified block diagram

Here is a modified block diagram of the setup used for collecting the address trace from the emulator board. Initially, we were using only the Output Enable (OE) signal from the emulator board (connected to the REQ1 and REQ2 pins on the PC data acquisition card) as a clock, due to which we were able to capture the instruction and load addresses. Now, Write Enable (WE) is also used which helps us capture the store addresses also. OE and WE are the two inputs to the NAND gate and the output is given as a clock to the REQ1 and REQ2 pins of the PC data acquisition card. This enables us to capture the instruction, load, and store addresses. The NAND gate operation was tested on the logic analyzer before implementing it in this setup.

In addition to NANDing OE and WE to form a single clock strobe for all addresses, we also sampled the two chip selects (SRAM and Flash), which are treated as additional address bits. The cache simulator can then examine the chip selects to correctly simulate instruction, data, and unified caches.

Note: The dashed lines in the diagram indicate the changes made.



Note: The dashed lines indicate the changes made.

## 11. Addendum: New experimental results

In this section, we present the experimental results obtained by performing experiments for various cache configurations. The configurations are similar to those used in Section 8. Direct mapped, 2-way set-associative, and 4-way set-associative caches are simulated. Two different cache block sizes are simulated, 16 bytes and 32 bytes. In addition to this, we provide separate results for unified caches, instruction caches, and data caches.

Each trace contains 1 billion addresses, which are gathered while the phone is in the idle mode (not processing a call).

### 11.1 Miss rate as a function of size and set-associativity

The graphs in Figures 10 (unified cache), 11 (instruction cache), and 12 (data cache) show miss rate as a function of cache size and set-associativity, for a block size of 16 bytes. Firstly, we observe that increasing the cache size reduces the miss rate. Instead of a smooth monotonically decreasing curve with diminishing returns in the end, we observe that for a direct mapped unified cache after 128KB there is a steep drop in the miss rate.

From Figure 10, the second observation is that increasing set-associativity from direct mapped to 2-way set-associative, for a unified cache, decreases miss rate substantially, from around 13% to 2% for a 32KB cache. The drop in miss rate is so large because a unified direct mapped cache naturally has a lot of conflict misses between instructions and data. Miss rate further decreases when set-associativity is increased from 2-way to 4-way, but the absolute reduction is less than before, matching our expectations of diminishing returns.

For the instruction and data cache (Figures 11 and 12, respectively), the delta between the miss rates for the direct mapped and 2-way set-associative caches remains fairly constant from 4KB to 32KB – a delta of about 0.7% and 1% respectively.

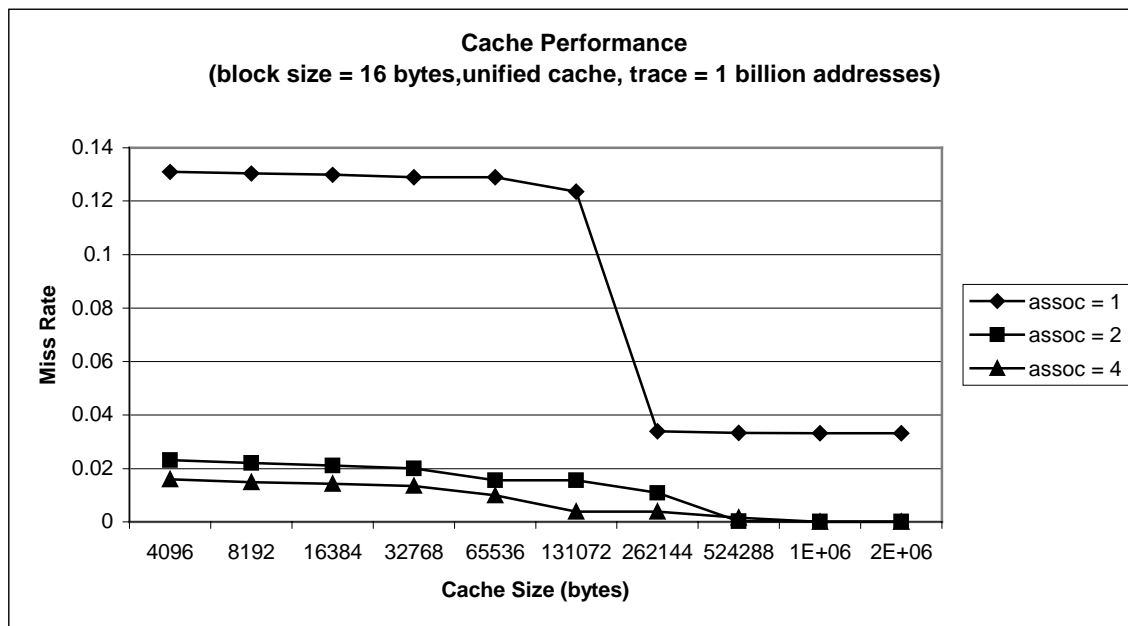
The graphs in Figures 13 (unified cache), 14 (instruction cache), and 15 (data cache) show the experimental results for a block size of 32 bytes. Note, all of Figures 10 through 15 use the same trace. The 32 byte block size shows the same trends as the 16 byte block size: larger cache size and higher set-associativity decrease miss rate, both show smooth monotonically decreasing miss rates with diminishing returns. The delta between the miss rates for the direct mapped and 2-way set-associative cache is very small for the instruction cache. However, for the data cache, the miss rate drops from 1.8%, for a direct mapped 4KB cache, to 0.8%, for a 2-way set-associative 4KB cache. The effect of blocksize on miss rate is explained in detail in Section 11.2.

We also performed a few “sanity checks” of the data in Figures 10 through 15. First, a 4-KB direct-mapped unified cache performs significantly worse than either a 4 KB direct-mapped instruction cache or a 4 KB direct-mapped data cache. And this

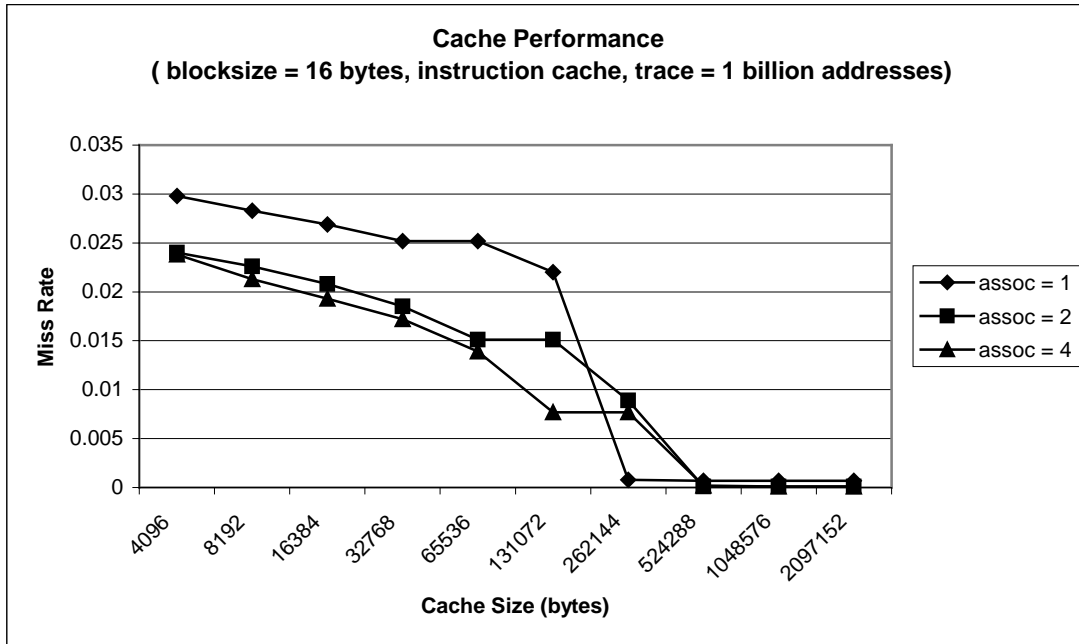
applies to the other cache sizes, as well. The reason is instructions and data have conflicts, and a direct-mapped cache is especially prone to conflict misses. E.g., for a direct-mapped 4KB cache with a blocksize of 16 bytes, instruction cache misses = 5,032,237, data cache misses = 19,997,054, unified cache misses = 130,818,630.

On the other hand, we know that *associative* unified caches are better at resolving conflicts between instructions and data. Therefore, the second sanity check is that we would expect the number of instruction misses plus the number of data cache misses to be close to the number of unified cache misses, for the associative caches. We confirmed this: Instruction cache misses = 2,909,698, data cache misses = 9,056,080, unified cache misses = 13,497,284, for 4-way, 32 KB caches with a blocksize of 16 bytes.

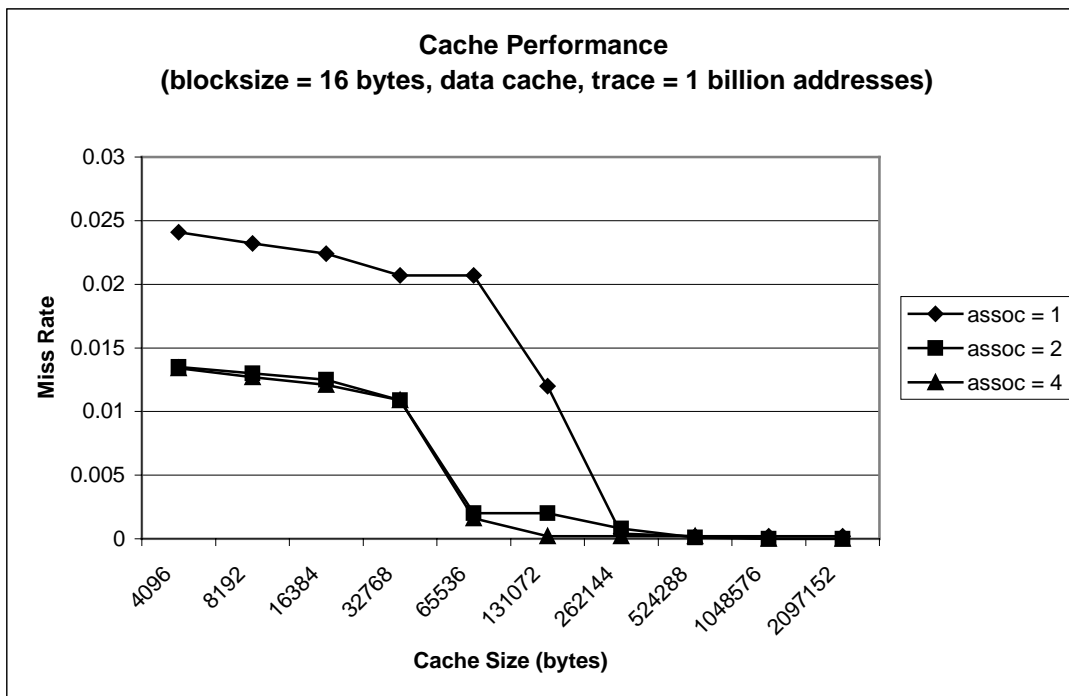
From Figure 11, the miss rate of a 256KB direct mapped instruction cache with a blocksize of 16 bytes is less than the miss rate of a 256KB 2-way (and even 4-way) set-associative instruction cache, whereas the same is not true for the corresponding unified and data caches.



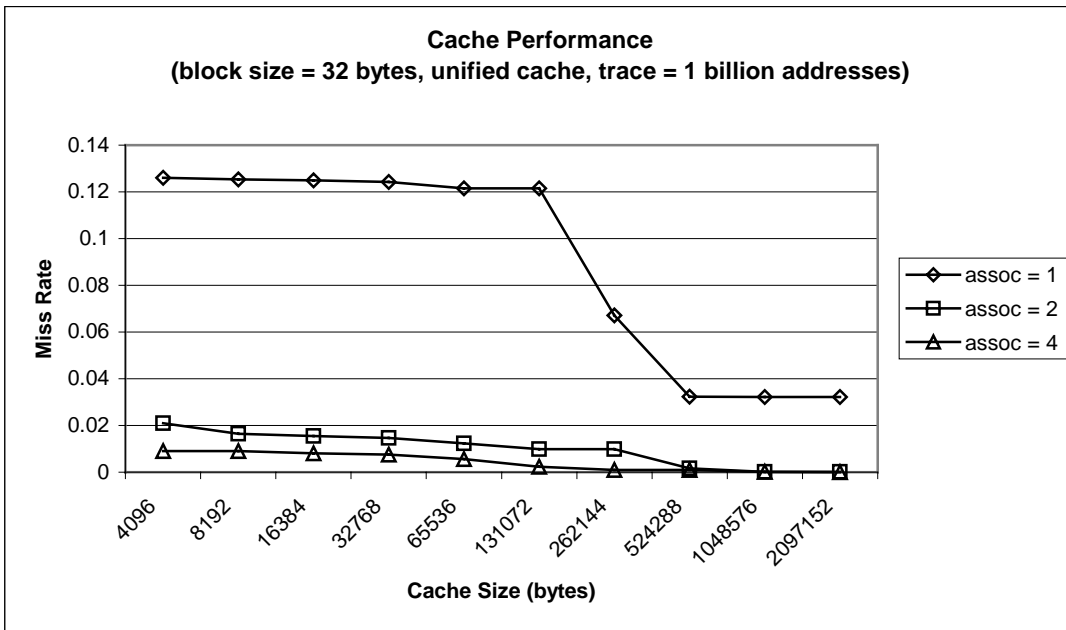
**Figure 10. Cache simulation for unified cache using 1 billion addresses. Block size is 16 bytes and cache size and set-associativity are varied.**



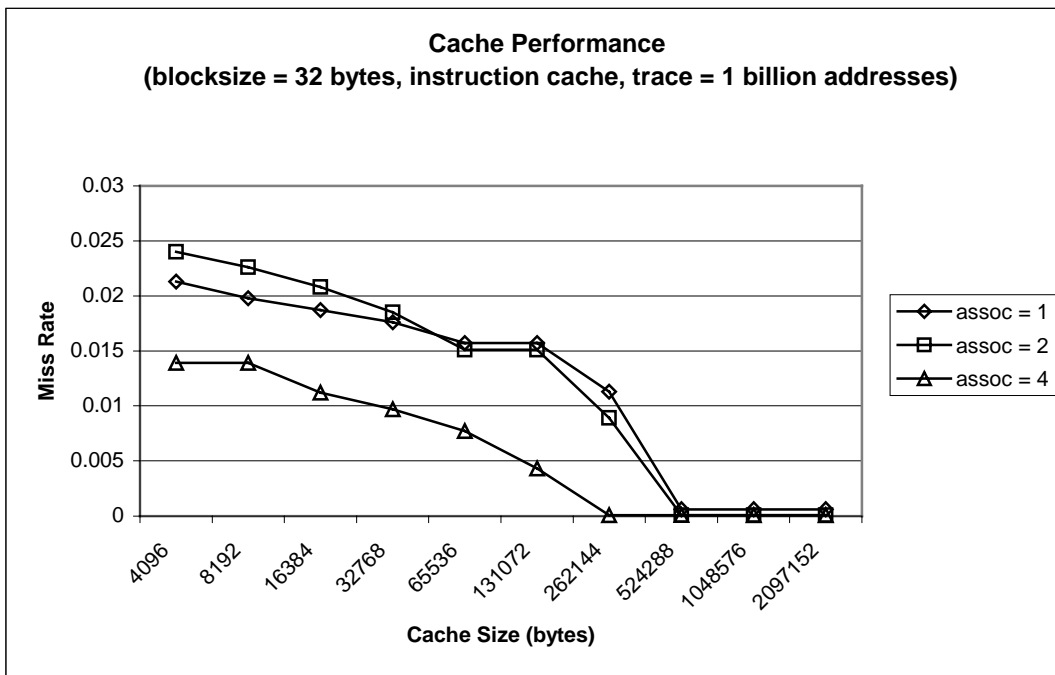
**Figure 11.** Cache simulation for instruction cache using 1 billion addresses. Block size is 16 bytes and cache size and set-associativity are varied.



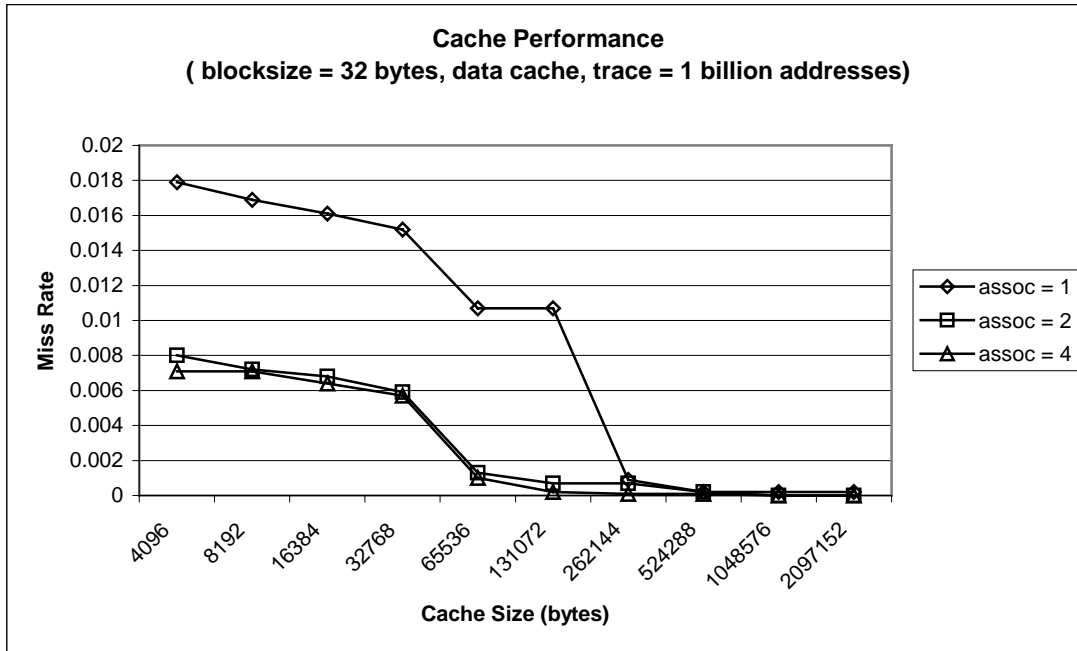
**Figure 12.** Cache simulation for data cache using 1 billion addresses. Block size is 16 bytes and cache size and set-associativity are varied.



**Figure 13. Cache simulation for unified cache using 1 billion addresses. Block size is 32 bytes and cache size and set-associativity are varied.**



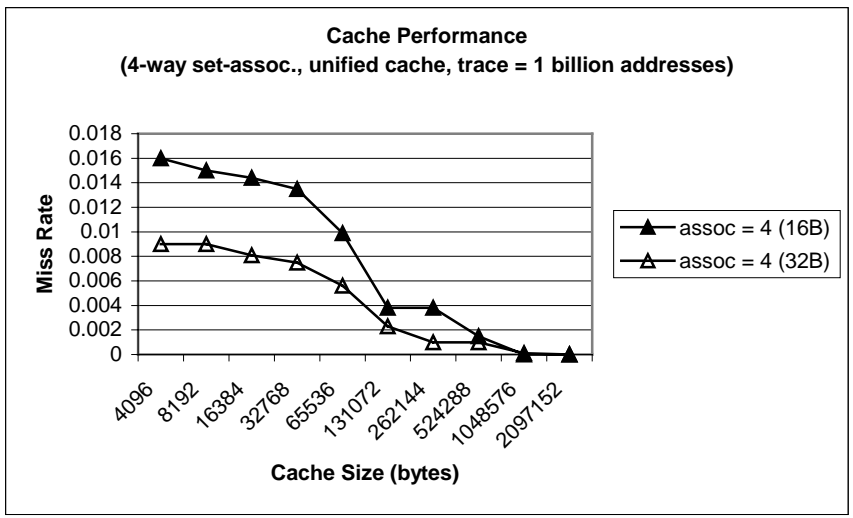
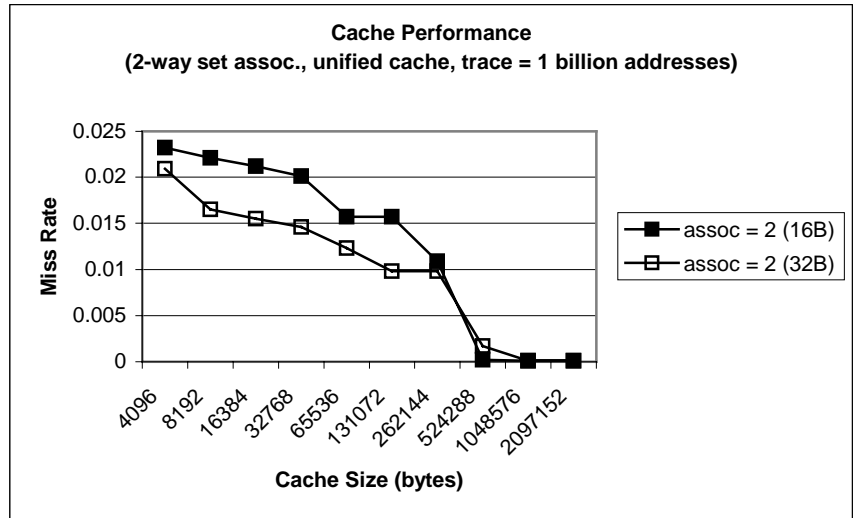
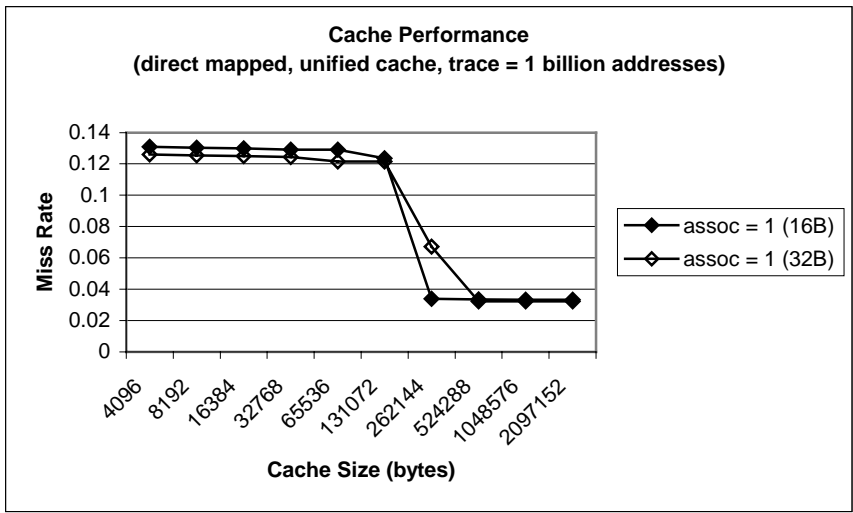
**Figure 14. Cache simulation for instruction cache using 1 billion addresses. Block size is 32 bytes and cache size and set-associativity are varied.**



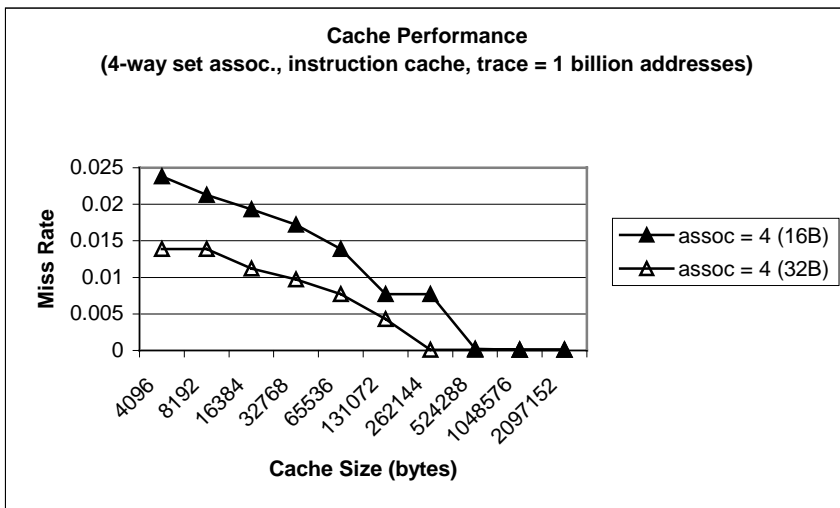
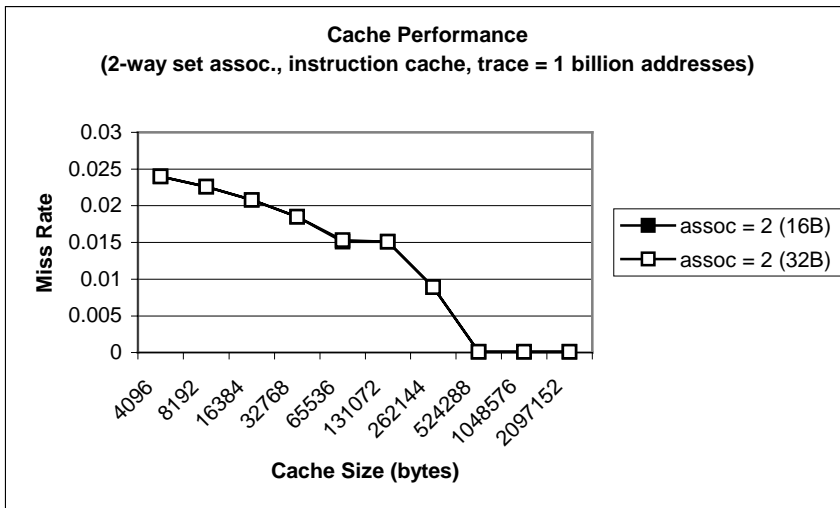
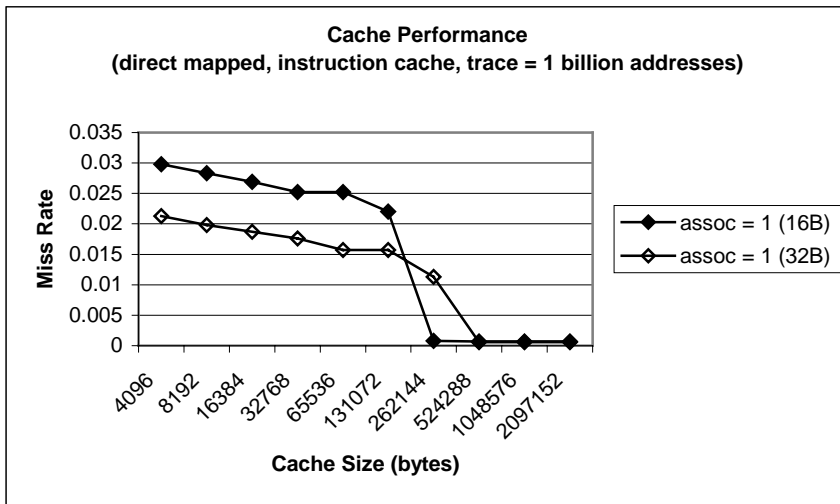
**Figure 15. Cache simulation for data caches using 1 billion addresses. Block size is 32 bytes and cache size and set-associativity are varied.**

### 11.2 Effect of block size on miss rate

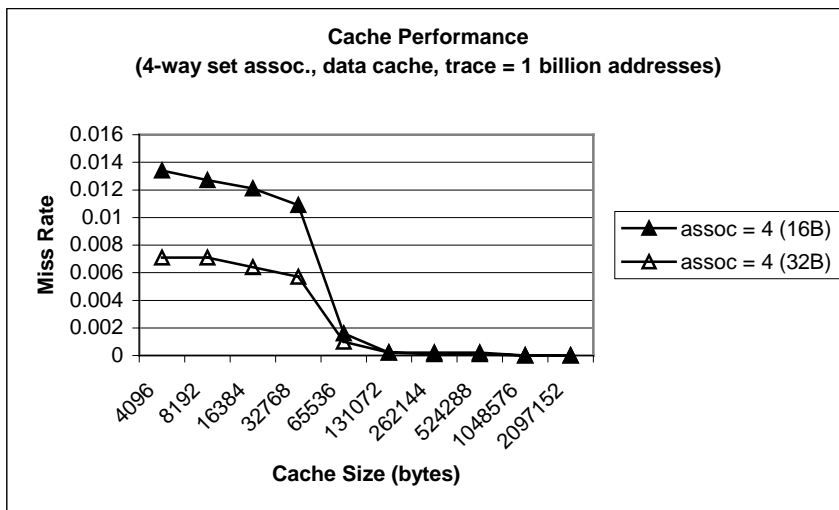
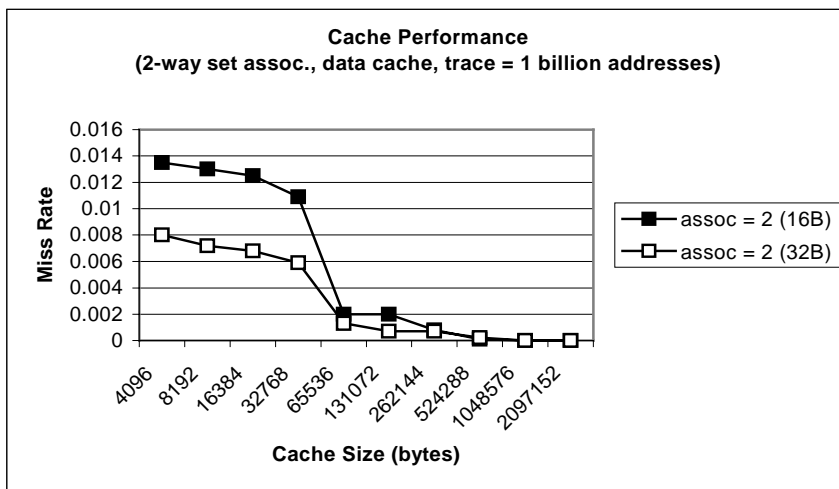
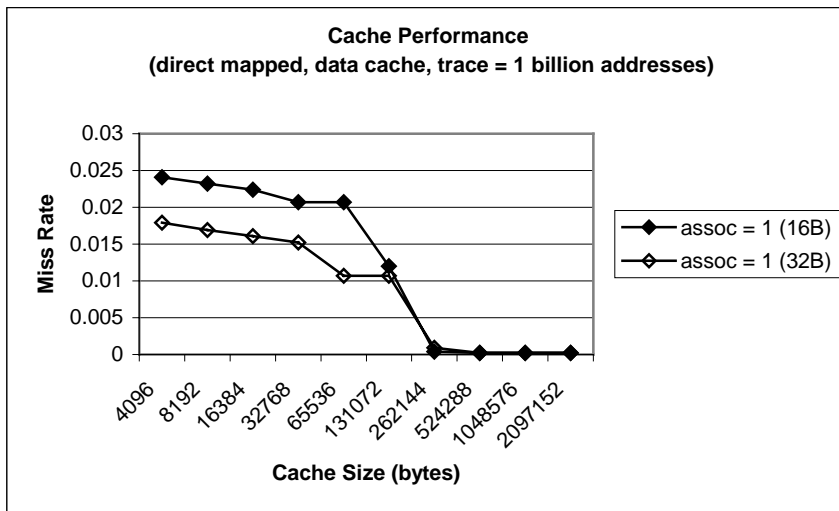
Figures 16 (unified cache), 17 (instruction cache), and 18 (data cache) show the effect of block size — 16 bytes (16B) versus 32 bytes (32B) — for each of direct mapped, 2-way set-associative, and 4-way set-associative caches. 32-byte block size usually performs better than 16-byte block size, presumably due to better spatial locality.



**Figure 16. Effect of block size (16B vs. 32B) is shown for each of direct mapped, 2-way set-associative, and 4-way set-associative unified caches.**



**Figure 17. Effect of block size (16B vs. 32B) is shown for each of direct mapped, 2-way set-associative, and 4-way set-associative instruction caches.**

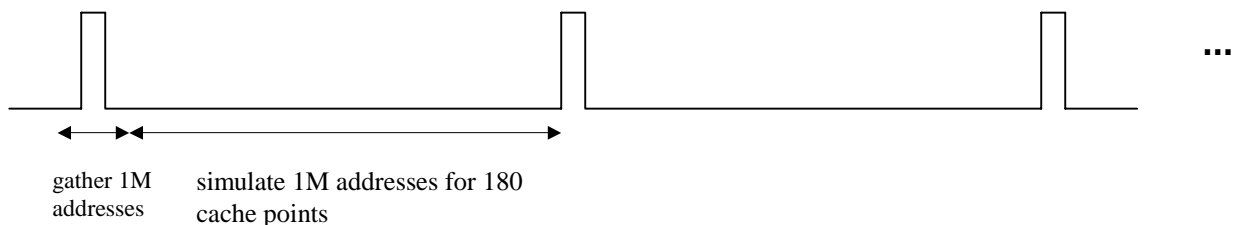


**Figure 18. Effect of block size (16B vs. 32B) is shown for each of direct mapped, 2-way set-associative, and 4-way set-associative data caches.**

Some interesting anomalies are observed from the graphs. First, from Figure 17, the miss rate of a 256KB direct mapped instruction cache, with a blocksize of 16 bytes, is less than the miss rate of a 256KB direct mapped cache, with a blocksize of 32 bytes.

Second, from Figure 14 (in the previous section), the miss rate of direct mapped instruction caches is less than the miss rate of 2-way set-associative instruction caches for 4KB, 8KB, 16KB, and 32KB instruction caches. This is for a line size of 32 bytes. This anomaly does not occur for 16-byte lines.

### 11.3 Why 1 billion addresses?



**Figure 19. Address sampling timing diagram.**

The cache simulator is embedded in the data acquisition code. The timing diagram of the process will look like the diagram in Figure 19 . The actual time of data acquisition is very small, compared to the time the simulator is running. This can be explained in the following way.

The time taken to collect and simulate a 100M long address trace was approximately 45 minutes (Note, we gathered miss rates for 180 different cache configurations at the same time! The machine is a 1 GHz Pentium-III). The data acquisition code is written such that 1M addresses are collected at a time and then simulated. This process is repeated 100 times. So, we can say that one iteration (data acquisition + simulation) takes 45/100 minutes, i.e., 27 seconds, to complete. The clock frequency of the emulator board at which the 1M samples are collected is approximately 10MHz. So the time required to collect 1M samples is approximately 0.1 second. Let's assume an upper limit of 1 second for collecting the addresses because the cell phone processor does more than memory bus operations. Thus, the simulation time equals,  $27 - 1 = 26$  seconds. So, out of the 27 seconds per iteration (i.e., per 1M addresses), 1 second is spent gathering the addresses, and 26 seconds are spent simulating. We miss a lot of addresses while simulating. The bottom line is we should gather as many samples as possible. Whether 1 billion addresses (i.e., 1000 samples) are statistically enough, we don't know for certain. Further experimentation and theoretical analysis are needed. However, we did increase samples 10-fold from the experiments in Section 8.

## **Appendix A: Source code for NI-DAQ trace gathering program**

## **Appendix B: Cache simulator source code**

## **Appendix C: Emulator board schematic**

Please refer to the separate PDF attachment for the emulator board schematic.

## **Appendix D: Directory structure for the application software, the cache simulator/tracer, and manuals**

```
C:\
|
|__ Ericsson
    |
    |__ DAQ
    |__ DAQdriver
    |__ data
    |__ for_the_presentation
    |__ Hashtablefiles
    |__ Installables
    |__ Labview
    |__ Manuals
    |__ Misc
    |__ Paul
    |__ Programs
    |__ traces_LA
```