an URI / NEU collaboration

Levo Machine

work status August 2000

student **David Morano**
advisors **Professor David Kaeli**
Professor Augustus Uht

NUCAR seminar 00/08/25

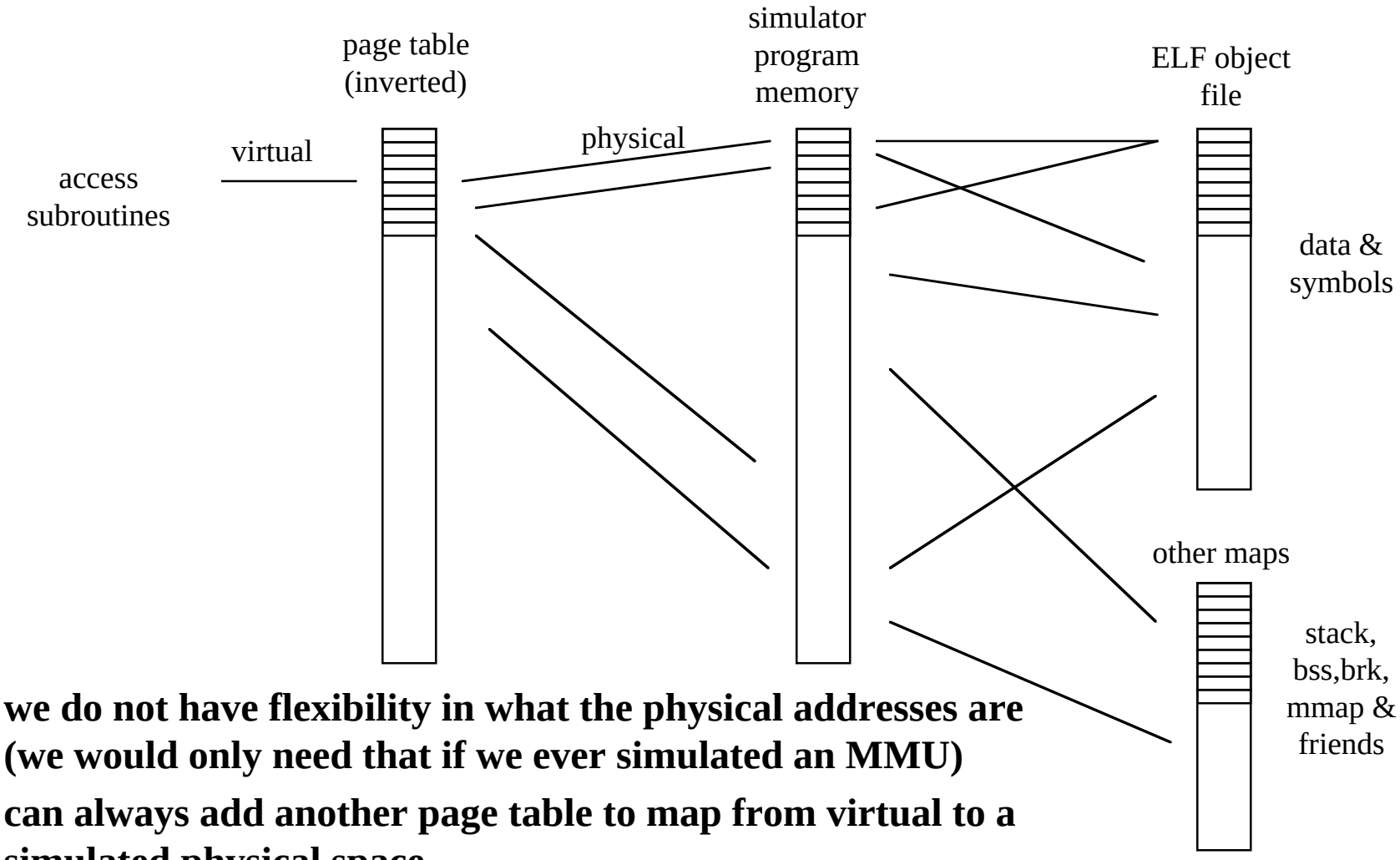
Agenda

- **background on program loading**
  - simulated program mapping
  - simulated program memory reading and writing
- **symbol table management so far**
- **system call identification**
- **system call handling (Who needs to do what ?)**

Program Loading

- handled by LMAPPROG (instantiated and called by LSIM with the program file, arguments, and environment)
- one program is loaded for each instance of LMAPPROG
- a program starts life as its absolute object file (ELF)
- all programs must be compiled for MIPS R3000 and STATICALLY linked ; we cannot handle dynamically linked programs (at this time)
- please do not strip the symbols from the program ; we need them for system call identification (allow for no symbols ??)
- programs are actually mapped instead of read in ; we mimic what the OS does at program load time ; this saves on swap space
- we also map program virtual address space for stack and other dynamic things in the future for some system call emulation
- we also map all symbol tables found in the program for general access

Program Loading (mapping)



- **we do not have flexibility in what the physical addresses are (we would only need that if we ever simulated an MMU)**
- **can always add another page table to map from virtual to a simulated physical space**

Program Mapping Inputs

- **we need the simulated program absolute object file**
  - currently set in the parameter 'paramtab' file
  - must be :
    - absolute executable
    - ELF object file
    - MIPS R3000
    - statically linked
    - a 32-bit program (not 64-bit)
    - MSB32 organized
    - other things
- **simulated program arguments**
  - currently set in the parameter 'paramtab' file
- **simulated program envionment**
  - currently defined in the configuration file 'levosim.conf' or 'conf'

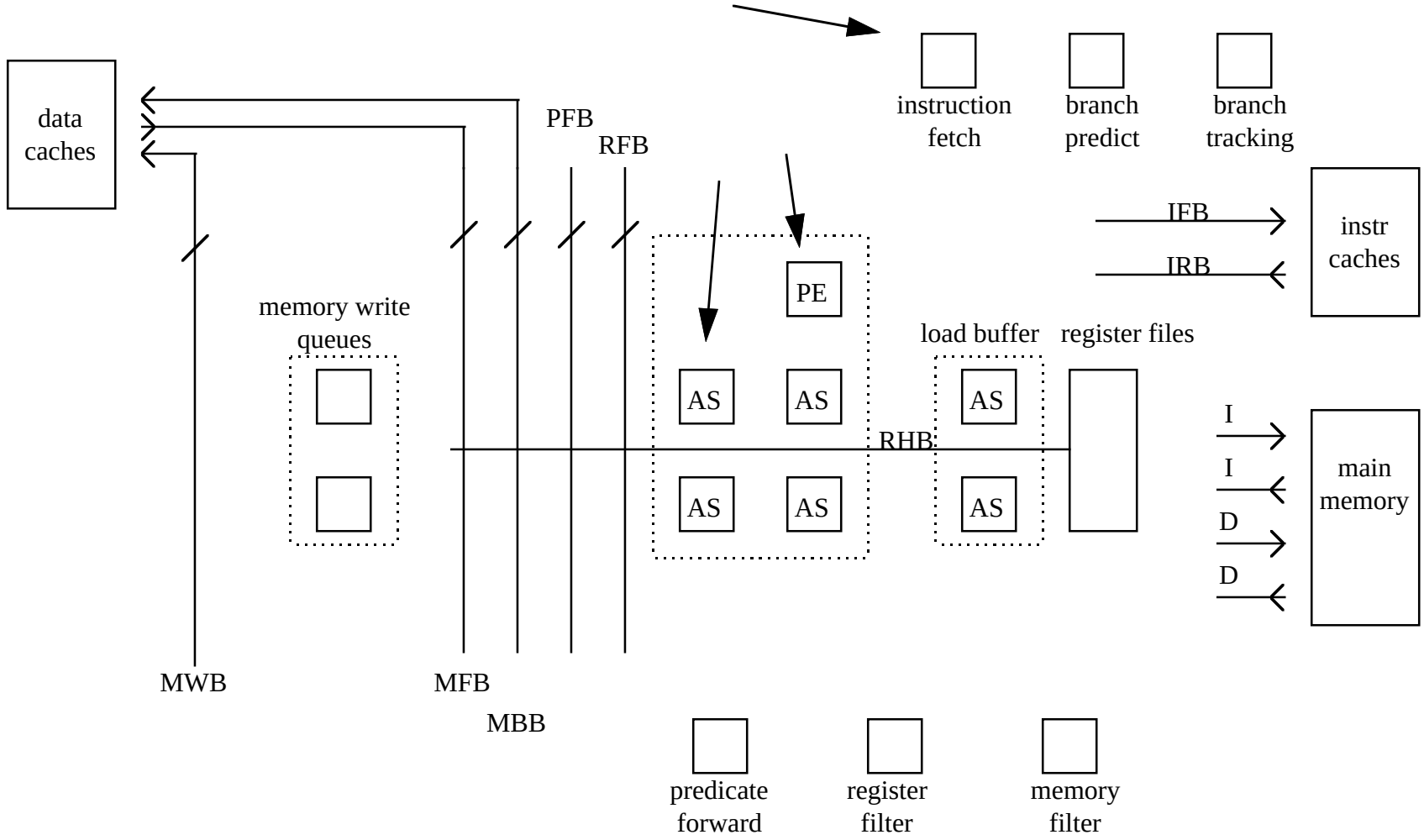
Program Mapped Information

- **currently the following information is made publically available from a LMAPPROG object :**
  - program BREAK address (for 'brk(2)' and 'sbrk(2)' system calls)
  - program entry address
  - initial program stack pointer
  - program symbols
  - interfaces for program memory access
- **the simulated program symbols from all symbol tables are indexed by symbol name and made available publicly through a name query interface**
- **I will probably make an enumeration interface to the symbols in the future also**
- **also availble but not as useful is an interface to convert virtual program addresses into physical (simulator program) addresses**

Program Memory Access

- **all access to program memory and system call identification is through the LSIM code code piece object (LSIM manages a LMAPPROG for our given simulated program)**
- **LSIM takes a list of system call subroutine names (supposedly that we are emulating) and looks them up in the LMAPPROG object**
- **an index of all system-call symbols is then made based on virtual address and this provides the basis for identifying system calls at run time**
- **current public subroutines include**
  - lsim_readint()
  - lsim_writeint()
    - writes anything (any byte combination) up to an 'int' size (4 bytes)
  - lsim_readinstr()
  - lsim_issyscall()
- **may provide a "closest symbol lower than value" function for easier debugging of simulated programs later**

Catching SYSCALLs



Who Needs to Do What ?

- **LIFETCH is currently using 'lsim_readinstr()' ; this will not always be the case ! ; we want to move this down to DRAM simulation at some point**
- **LIFETCH needs to check fetch addresses to see if they stray into a SYSCALL subroutine ('libc(2)')**
- **LIFETCH should not continue fetching into a 'libc(2) -- SYSCALL' subroutine because that code will never be executed by the Levo machine**
- **LIFETCH should fetch PAST a pseudo SYSCALL ! (that is where the execution flow will be)**
- **LIFETCH cannot really identify a SYSCALL call since they are all done through 'jmp and link' instructions and LIFETCH has no knowledge of Levo architected registers !**

Who Needs to Do What ?

- **LAS should recognize a SYSCALL call by the EA of the jump**
- **there could be some involvement with the LPE for handling SYSCALLs if the LPE is needed for a jump EA calculation**
- **LAS should NOT attempt to actually ever execute a SYSCALL (real or pseudo)**
- **if a pseudo SYSCALL ever makes it to "about to be committed" then the LAS code should take a simulator trap to emulate the pseudo SYSCALL**

Who Needs to Do What ?

- **IW code must recognize that there are SYSCALLs in the i-window and arrange to suspend execution on paths that have SYSCALLs in them**
- **real SYSCALLs might end up in the i-window also (they belong to a system call that we have not caught or can handle)**
- **if a real SYSCALL ever makes it to "about to be committed" then ALL simulated program execution must stop due to a non-supported system call**
- **if a pseudo SYSCALL ever reaches the "about to be committed" state (will happen when we execute real programs) then we have to trap out of the Levo machine and emulate that SYSCALL**