CSE 661 – Parallel and Vector Architectures

Main Research Project

Executing Parallel Loops on a Multicore Processor

Due Sunday, January 20, 2008 by 11 Midnight

Problem Statement

Current mainstream microprocessors provide limited support to data parallelism. In 1997, Intel introduced MMX instructions in the Pentium II processor to accelerate the performance of multimedia and communication applications. These instructions define a simple SIMD execution model and operate on 64-bit registers, which can pack bytes, 16-bit words, or 32-bit double words. In 1999, Intel introduced SSE instructions that operate on 128-bit XMM registers, which pack floating-point data. Since then, Intel continued to extend and improve its SSE technology by introducing new SSE2/3/4 instructions that operate on a variety of packed integer and floating-point data. Other microprocessor manufacturers also extended their instruction set architecture to support data-level parallelism within 128-bit registers to accelerate media-rich applications that include audio and video compression, 2D image processing, 3D graphics, speech recognition, and signal processing for communication. Sun enhanced the SPARC processor with VIS, Hewlett-Packard added MAX to the PA-RISC architecture, and IBM/Apple/Motorola added AltiVec to the PowerPC.

The main drawback of the SIMD instructions is that they operate on fixed-size 128-bit registers, which limit the data-level parallelism that can be exploited. Today, chip multiprocessors integrate multiple cores. The data-level parallelism provided by SIMD instructions is not scalable and does not take advantage of the additional resources provided in the multiple cores. To scale parallelism, one has to create multiple threads to run in parallel in the multiple cores. Thread creation is done at the operating system level, which adds overhead and complexity to the execution of parallel loops.

The main idea of this research project is to enable a parallel loop to execute across all cores, taking advantage of the multiplicity of resources provided in a multicore processor. With the introduction of few instructions and minimal extra hardware support, a thread running in a single core will be able to broadcast a parallel loop to all cores. This mode of execution will be supported completely in the microarchitecture. The operating system "sees" the parallel loop as one thread, rather than multiple threads running on multiple cores.

A chip multiprocessor with *N* cores and a shared L2-cache is depicted in Figure 1. Each core is capable of executing a thread scheduled by the operating system, referred here as a *root* thread, plus few additional threads created by the hardware to speedup the execution of parallel loops. An L2 cache is shared by all the cores. To optimize the bandwidth and latency, the L2 cache is divided into *M* independent banks that operate in parallel. The *N* cores communicate with the *M* cache banks using the on-chip interconnect. There is nothing new about Figure 1, except the ability to execute parallel loop instructions across all cores. A parallel loop instruction is *effectively converted into N scalar instructions* by distributing its work on the *N* CPU cores.

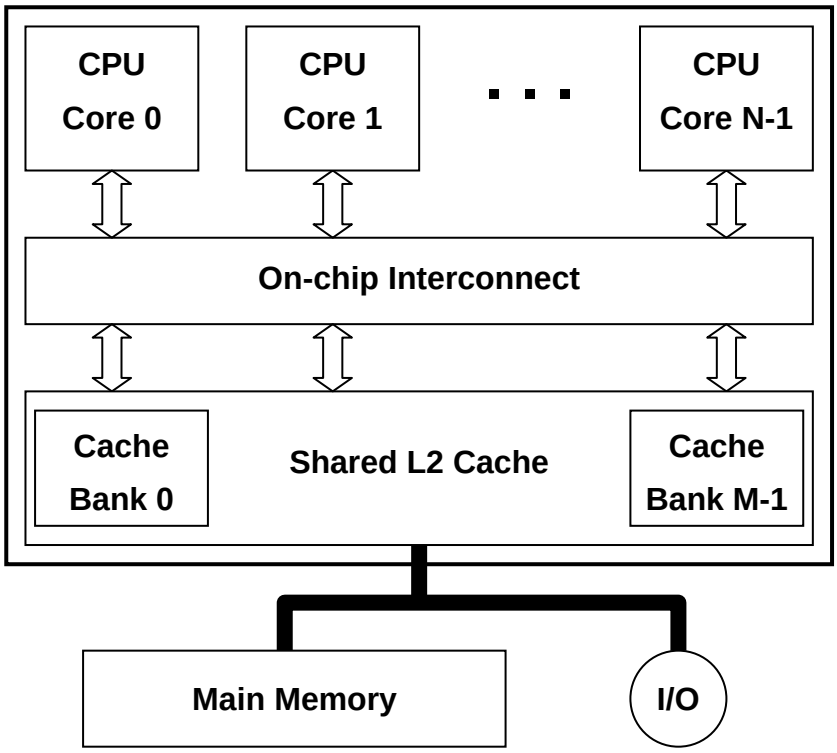

Figure 1: Chip multiprocessor with *N* cores and a shared L2 cache with *M* banks

Parallel Loops

Consider the execution of the following loop where *x* and *y* are vectors residing in memory and *a* is a scalar value. This is the DAXPY loop that forms the inner loop of the Linpack benchmark for performing Gaussian elimination:

```
for (i=0; i<n; i++) y[i] = a * x[i] + y[i];
```

The iterations of the above loop can be executed in parallel. Assume that the base addresses of arrays *x* and *y* are in registers **r2** and **r3** respectively, and the scalar values *n* and *a* are loaded in registers **r1** and **f1** respectively. Then, the above loop can be translated as follows:

```
   vp      r1, L2          ; allocate virtual processors
   dup     f1 = f1         ; duplicate f1 in all VPs
   dup     r1 = r1         ; duplicate r1 in all VPs
L1:
   lv      f2 = (r2+)      ; load vector x into f2 in all VPs
   lv      f3 = (r3)       ; load vector y into f3 in all VPs
   fmul    f4 = f1, f2     ; multiply: a * x[i] in all VPs
   fadd    f4 = f4, f3     ; add: a * x[i] + y[i] in all VPs
   sv      (r3+) = f4      ; store vector f4 at address y
   loop    L1
L2:
```

The VP Instruction

The above loop appears to be sequential, but is in fact a parallel loop. The **vp** (Virtual Processor) instruction allocates virtual processors to execute a parallel loop. Each virtual processor is a hardware context that includes integer and floating-point register files, a program counter, and some additional control registers. The **vp** instruction specifies the vector length and the label address at which to terminate parallel execution. The **vp** instruction enables the exploitation of data-level parallelism across multiple cores.

In addition to allocating hardware contexts across all cores, the **vp** instruction initializes the vector length (VL) register in all hardware contexts with the specified number of iterations, the virtual processor identification (VPID) register with a unique number, the virtual processor count (NVP) register with the count of VPs (a power of 2), and the Root register with the root core number. The **vp** instruction also initializes the end program counter (EPC) register in all the allocated VP contexts with the label address that marks the end of the parallel loop, and the program counter (PC) registers with the address of next instruction to launch parallel loop execution. This is illustrated in Figure 2, where Core 1 is the root core that issued the execution of the **vp** instruction. The root core always have VPID = 0. The instructions appearing after **vp** will be executed as asynchronous parallel threads (not in lockstep) on all the virtual processors until the end label is reached. This mode of execution is more flexible than the lockstep vector execution mode implemented in vector processors.

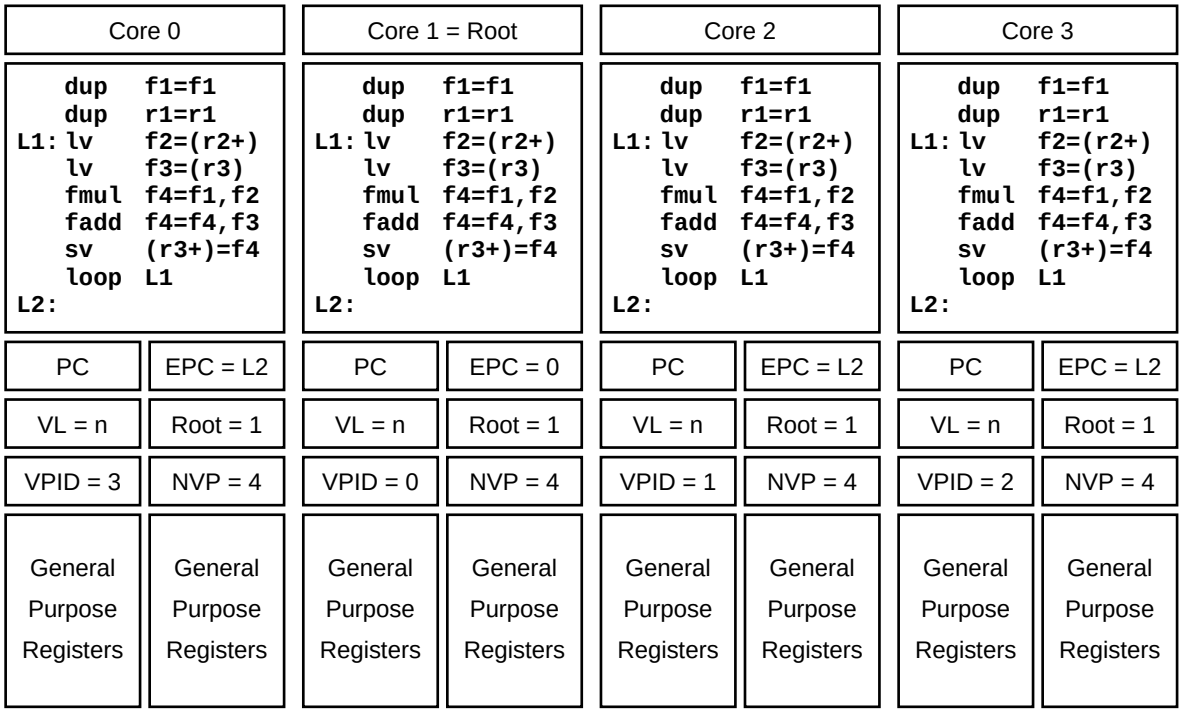

Figure 2: Executing a parallel loop on four cores

When the program counter (PC) reaches the end label program counter (EPC), the virtual processor terminates execution and the hardware context is freed. Eventually all virtual processors will free their hardware context, except for the root thread, which continues normal execution after the end of the parallel loop. A special case occurs when the vector length is equal to 0. In this case, no virtual processor is allocated and the **vp** instruction simply becomes a jump to the end label, skipping all instructions in a parallel loop.

If all hardware contexts are allocated and the **vp** instruction fails to allocate new ones, then the parallel loop will be executed sequentially in the root core, rather than as parallel threads in virtual processors.

The DUP Instruction

The **dup** (duplicate) instruction broadcasts a source register from the root thread to a destination register in the root and all the allocate VP threads. It is illustrated in Figure 3. The **dup** instruction accomplishes register-to-register communication. It acts as a non-blocking send and injects a token into the network when issued by a root thread. It acts as a blocking-receive in all the allocated VP threads. The root thread also copies the source register value into the destination register.

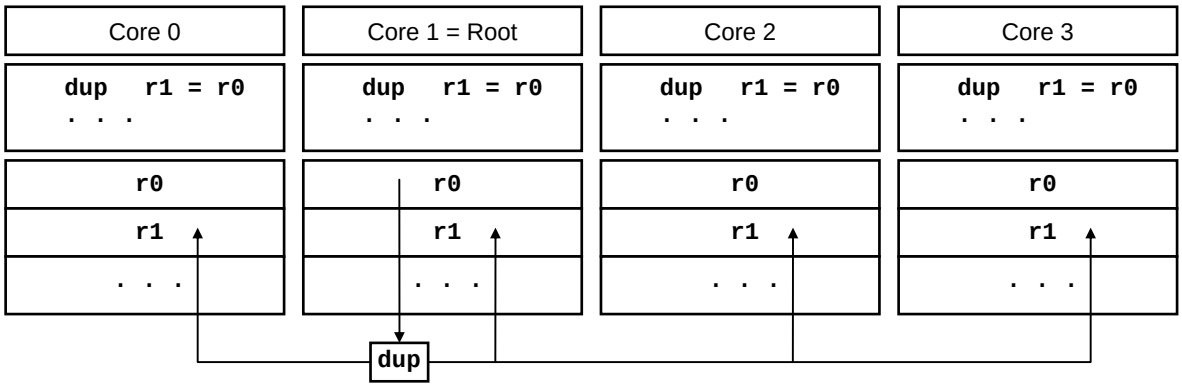

Figure 3: Broadcasting a source register to all VP threads, including Root

If no VP context is allocated, the **dup** instruction becomes a simple register-to-register move instruction within the root context. The **dup** instruction is used to duplicate integer as well as floating-point registers.

The LOOP Instruction

The **loop** instruction is used for executing a counter-controlled loop. It does the following:

```
if (VL > NVP + VPID) { VL = VL – NVP; jump label; }
else VL = 0;
```

Suppose VL = 10 and NVP = 4. After executing the first iteration, VL becomes 6 and the loop instruction jumps to label to start the second iteration in all virtual processors. After executing the second iteration, VL becomes 2 and the loop instruction starts a third iteration in the first two virtual processors (VPID = 0 and 1), while the loop terminates (VL = 0) in the last two virtual processors (VPID = 2 and 3). Therefore, three iterations are executed in the first two virtual processors (VPID = 0 and 1) and two iterations are executed in the last two virtual processors (VPID = 2 and 3). This is equivalent to executing ten iterations if the loop is executed only by the root thread. The **loop** instruction guarantees that VL = 0 in all virtual processors when the loop terminates.

The LV Instruction

The **`lv`** (load vector) instruction loads a contiguous block of memory from the shared L2 cache and distributes the words onto the virtual processor contexts. Each VP operates on one element of the vector. The **`lv`** instruction is issued only by the root VP. The other VPs act as receivers. This is shown in Figure 4. The **`(r2+)`** notation indicates that the address in **`r2`** should be updated: **`r2 = r2 + NVP×8`**. Alternatively, each VP can load its value by using a scalar load instruction.

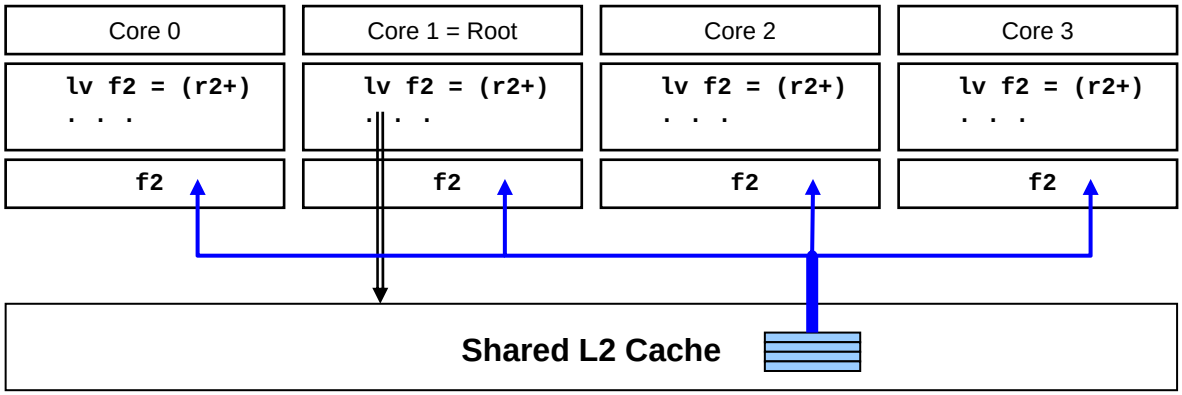

Figure 4: Load Vector done by the Root VP

More details on the instruction set and the processor architecture will be provided later.

Simulator Development

Develop a simulator that will simulate the functionality and performance of a multicore processor that has the capability of executing a parallel loop on multiple cores. Your simulator should be cycle-accurate and parameterized. It should produce performance statistics by directly executing benchmark programs. Search the Internet to find open-source simulators that might simplify your task. Do the necessary modifications to add the necessary features that you want to simulate in the proposed architecture.

Benchmark Program Kernels

Decide on the benchmark programs or loop kernels that you to simulate. You can write and simulate small loop kernels, such as matrix multiplication, Gaussian elimination, FFT, and some programs from the EEMBC benchmark.

Project Paper and Presentation

The final report should be written much like a journal or a conference paper. It should include a title, the names of the group members, an abstract, an introduction, a body describing details of the processor architecture, description of the simulator, simulation results, a conclusion, and a list of references. The abstract should summarize the accomplishments and contributions of your project in one or two paragraphs. The body should be single-spaced, 10-point font size, and of length 10-12 pages. Additional supporting material on the simulator development can be put in a separate appendix (not part of the paper).

A presentation will be given by each group on the due date during the last week of the semester. All group members should deliver part of the talk. Your talk should typically

include a title slide, an insight slide discussing the main idea of your work, an outline, a problem description, motivation, background, details of the processor architecture, description of the simulation, benchmark programs, results, conclusions, and possible future work and enhancements.

Submission Guidelines

All submissions will be done through WebCT.

Submit one zip file containing the source and executable code of the simulator, benchmark programs that were tested, the project paper, and the final presentation.