Report No. UIUCDCS-R-2005-2633 **UILU-ENG-2005-1823**

A Brief Description of the NMP ISA and Benchmarks

by

Mingliang Wei, Marc Snir, Josep Torrellas, and R. Brett Tremaine

February 2005

A Brief Description of the NMP ISA and Benchmarks*

Mingliang Wei+, Marc Snir+, Josep Torrellas+ and R. Brett Tremaine‡

+Department of Computer Science
University of Illinois at Urbana-Champaign
Thomas M. Siebel Center for Computer Science
201 N. Goodwin, Urbana, IL 61801-2302, USA
{mwei1, snir,torrellas}@cs.uiuc.edu

‡IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218, Yorktown Heights
New York 10598, USA
afton@us.ibm.com

February 2005

Abstract

The Near Memory Processor (NMP) is a multithreaded vector processor integrated with the memory controller. The NMP works subordinately upon requests from the main processors. The NMP is complementary to the conventional superscalar processors and it is optimized for the bandwidth bounded applications and bit manipulation workloads. A program addressable memory in the NMP, Scratchpad provides an effectively large register set to hold vectors, streams and frequently accessed values. Avoiding saving and restoring the vector registers during context switch, the scratchpad reduces the overhead of the multithreading and enables a simple NMP architectural design. We design an instruction set that includes vector, streaming and bit manipulation instructions. We present the instruction set architecture of the NMP in this report, including register sets, addressing mode and instruction formats. A brief description of the benchmarks is also included.

1 The Near-Memory Processor

This doucument briefly describes the ISA of the Near-Memory Processor[3]. Figure 1 presents the NMP in the base line environment of a shared memory multiprocessor. Each memory controller is integrated with one NMP. All address space, including the NMP storage (scratchpad) is shared and can be accessed both by the main (commodity) processors and by the NMPs.

The threads that the NMPs execute are called *Memory Threads* (MT). An MT context includes a set of registers that are described below. Not all these registers need to be saved and restored on context switching.

*This work is supported by DARPA Contract NBCHC-02-0056 and NBCH30390004.

2 General Purpose Registers

A small number (e.g., 32) of 64-bit general purpose scalar registers are available to each thread in the NMP. These registers store scalars and specifiers, where specifiers are used to refer to vectors or streams in the scratchpad.

3 Control Registers

They include Instruction Pointer Register, status registers and some other control registers. The mask registers which are used for conditional vector instructions are not provided. Each element in the vector, however, has an extra bit to hold the mask bit. The execution of a vector instruction produces no effect on the elements in the destination vector whose correspondent mask bits are 0s. Note, both the vectors and masks are not part of the context, they are stored in program addressable space (scratchpad).

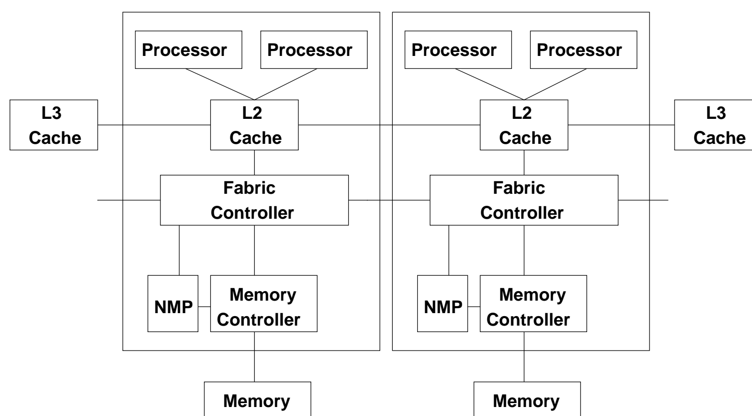

Figure 1: NMPs in a system like the IBM Power 5

4 Bit Matrix Register

(BMR) is a 64×64-bit register. The BMR is used for the *bmm* instruction [1] to permute the bits within a word. To permute a word W in a register, a bmm instruction bit-multiplies the BMR with W, the output of which is stored in the destination register. The bmm instruction enables efficient execution of various functions such as bit permutation, bit matrix transposition, column parity calculation, etc.

5 Vectors and Stream Specifiers

The data structures that refer to vectors or to stream buffers are called *vector specifiers* or *stream buffer specifiers*. They provide detailed information on how the vectors or stream buffers are stored. To access a vector or a stream buffer, the specifier which is loaded into the scalar register by previous

instruction is read, then the data are accessed via indirect mode.

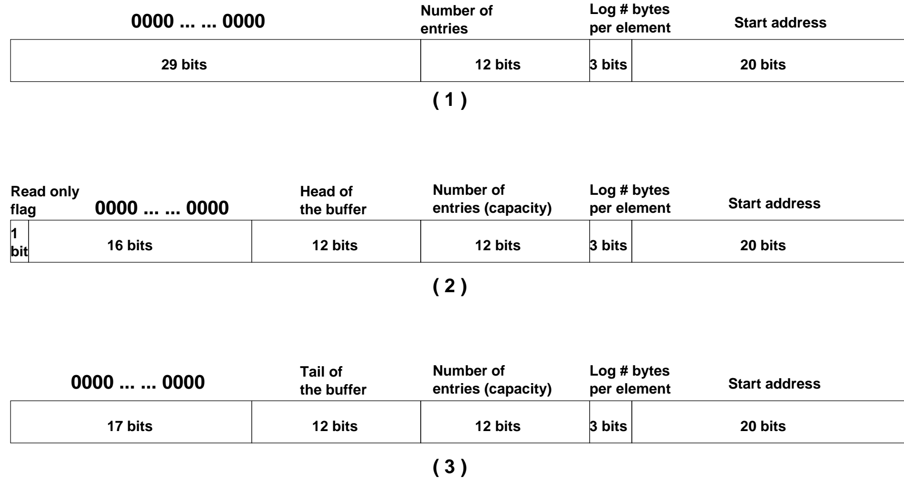

Figure 2: (1) Vector Specifier, (2) Stream Buffer Consumer Specifier and (3) Stream Buffer Producer Specifier.

Figure 2 describes the format of vector and stream specifiers. The structure for a vector specifier includes the scratchpad address for the first element, the number of bytes per element (logarithmic scale), specified as 2^{k+1}, where $0 \leq k \leq 7$, and the number of elements in the vector, specified as $k+1$, where $0 \leq k \leq 4095$. The elements in a vector are stored consecutively in the scratchpad.

There are two types of stream buffer specifiers: stream buffer consumer specifier and stream buffer producer specifer. The consumer specifier is used by the consumer of the buffer to peek (if the read only flag is 1) or deQueue (if the read only flag is 0) elements from the buffer. The producer specifier is for the producer to deposit elements into the tail of the queue (buffer). Both specifiers include the buffer starting address, the number of bytes per element (logarithmic scale) and the number of entries in the the buffer, i.e., buffer capacity. The difference is that the consumer specifier also contains the current head of the buffer, while the producer specifier contains the current tail of the buffer. Each stream buffer only has at most one producer and one consumer. The head and tail wraps around at the boundaries of the buffer.

6 Instruction format

The NMP has 32-bit instructions. New op-codes are added for new arithmetic and logical operations, e.g., *bmm*, *leadz*, etc. An NMP arithmetic/logic instruction has two source operands and one destination operand. All operands have to be either in the registers or in the scratchpad. Vector or stream instructions are identified by the addressing mode field in the instruction. *Load/Store* instruction moves data between the memory and the registers or the scratchpad. The scratchpad is treated as a register extension. Data movement between the registers and the scratchpad can be done via arithmetic instructions.

7 Addressing mode

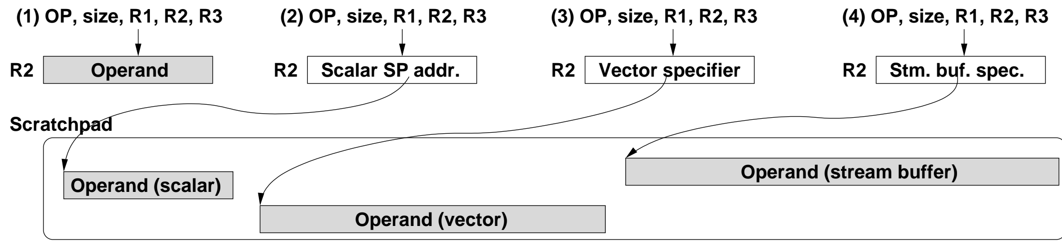

Figure 3: Addressing modes for NMP instructions. (1) Direct mode, (2) SCA mode, (3) VEC mode and (4) STM mode.

With operands possibly in the scratchpad, extra field in the instruction is required to specify the type, location and structure of the operands. Another field "size" is used to specify the number of elements that are involved. We provide an addressing mode field for this purpose. An operand can be of one of the following three types: a scalar, a vector or a stream buffer. Scalars can be stored either in the registers or in the scratchpad. Vectors and stream buffers can only be stored in the scratchpad. Therefore, each operand can be specified in one of the following four addressing modes: direct mode, scalar indirect mode (SCA), vector indirect mode (VEC) and stream buffer indirect mode (STM). The STM mode could possibly be further divided into stream buffer producer mode and stream buffer consumer mode. We did not do so, because these two STM modes can be determined by whether they are of source or destination operands. A source operand can only be of consumer mode; a destination operand can only be of producer mode.

Operands in the registers are accessed in direct mode. Operands in the scratchpad are accessed indirectly (see Figure 3). In the first case, the register indicated by the instruction contains the operand; in the next three cases, it contains an scratchpad address (SCA), a vector descriptor (VEC) or a stream buffer descriptor (STM) respectively. Note, some of the combinations of the three addressing modes (two for the source operands and one for the destination operand) in one instruction may not be valid, so the addressing mode field can actually be encoded in less than 6 bits. In fact, an alternative, we could use only one bit for the addressing mode of each operand, direct or indirect, and encode the remaining info in the referred registers.

8 Bit Manipulation Instructions

Table 1 summarizes the bit manipulation instructions that we propose.

9 Vector Instructions

Each scalar arithmetic or logic instruction has a vector counterpart which applies the same operation on every element in the vector. Load/store moves data between the memory and the registers or the scratchpad. Sequential, strided and indexed (scatter/gather) main memory access patterns are

Instruction	Remarks
Leadz	Count the leading zeros of a scalar.
Popcnt	Count the number of ones in a scalar.
Bmm_load	Load the 64×64-bit matrix from the scratchpad into the BMR. It is a special vector load instruction (A regular vector load instruction transfers data between the scratchpad and the main memory.).
Bmm	Bit multiply the source operand with the matrix in the BMR. For $bmm(s_i, s_k)$, each bit j of the 64-bit integer result s_i, counting from the highest order bit position down to the lowest, is computed thus: $s_{ij} = popcnt(s_k \& bmr_j)$ (mod 2), where bmr_j is the jth row of the BMR [1].
Sshift	Logic left- or right-shift a block of data, e.g., 128 bytes. The shift can be rotational or not rotational. In the latter case, zeros are shift into the block.
Mix	Bit-interleave higher(lower) half of two words.

Table 1: Bit Manipulation Instructions

Application	Vector?	Stream?	Bit Manip?	# Threads	Remarks
Rgb2yuv	X			4	Convert the RGB presentation to YUV
ConvEnc	X	X	X	3	Convolutional encoder
BMT			X	4	Bit matrix transposition
BSM		X	X	3	Bit stream manipulation
3DES	X		X	4	3DES encryption
PartRadio	X	X		3	Partial radio station
Stream	X			4	Simple vector operations

Table 2: Applications evaluated.

supported. Vector element size can be smaller than one word so that vector loads and stores can be used for subword permutations. For the instructions that set masks for a later vector operation, the destination vector of the later vector operation is given as an operand of the mask-setting instruction. The mask bits of correspondent elements in the destination vector are set, which will be used in the later vector operation w.r.t. the mask.

10 Operations on Stream Buffers

No special instructions are provided to operate stream buffers. Instructions use STM mode to refer to stream buffers. The specifiers for the stream buffers are updated implicitly when elements enter or leave the buffer. Full/Empty bits are used in the buffer to detect if the buffer is full or empty. If the head of the buffer is empty, the consumer has to wait. The producer has to wait until the tail of the buffer is empty to deposit new elements.

11 Benchmarks

For the evaluation, we select a number of small applications that we list in Table 2. On average, the applications have 730 lines of C code. The table shows if the applications can be vectorized, use streams, or use bit manipulation instructions. The table also shows the number of concurrent NMP threads used for each application.

To understand the NMP benchmarks, we briefly describe what they do.

Rgb2yuv converts an image (1000 × 200 pixels) in RGB color format to YUV color format. We

execute four threads concurrently. Each thread processes part of the input data stream.

ConvEnc performs a convolutional encoder algorithm, which adds redundancy to a binary stream for forward error correction. A binary, half rate (2 bits of output for each input bit) bit stream is encoded with a constraint length of 3. The generating polynomials that we use are $G_0 = 1 + D_1 + D_2$ and $G_1 = 1 + D_2$. The input stream is 375K bytes.

For *ConvEnc*, we generate three threads: one thread reads the binary stream to be encoded into a stream buffer in the scratchpad; a second thread performs the encoding, processing a 64-byte block at a time, and stores the results into another stream buffer; the third thread takes the results from the stream buffer and writes them back into the memory. We use vector operations, a block shift instruction *Sshift*, and a bit manipulation instruction *Mix*.

BMT tests the bit manipulation ability of the NMP. The input is a binary stream (about 4M bits). Each consecutive 1024×1024 bits in the stream are treated as a bit matrix. The bit matrices are transposed and the resulting matrices are stored back to the memory. We use four threads, each of which works on a partition of the input data. In each thread, the 1024×1024 bit matrices are divided into 64×64 submatrices, and the Bmm instruction is used to transpose the 64×64-bit submatrices.

BSM manipulates a binary stream. The stream is first split into two streams. Then a new binary stream is computed based on those two. Finally, we identify sequences of zero runs in the stream. For each sequence identified, we output the starting position and the length of the sequence. These operations are performed with three threads (*generator*, *splitter* and *counter*). The *generator* generates the first bit stream and deposits it into a stream buffer in the scratchpad. *Splitter* splits it and deposits the two resulting streams into two stream buffers. *Splitter* also computes an intermediate stream, which is also stored in the scratchpad. The *counter* takes elements from the three stream buffers, calculates the final stream and calculate statistics on zero runs. The input stream is 1M bits.

3DES performs 3DES encryption in counter mode for 80k bytes. Four threads are used, each of which works on a partition of the input data.

PartRadio is an FM radio with multi-band equalizer. The input (10k floating point numbers) passes through a demodulator to produce an audio signal, and then an equalizer. We use three pipelined threads: a low pass filter, then a demodulator, and then an equalizer.

Stream [2] is a simple synthetic benchmark program that measures sustainable memory bandwidth and the corresponding computation rate for simple vector kernels. The benchmark evaluates the performance of four simple vector kernels: Copy, Scale, Add and Triad. On the NMP, we run four threads in parallel, each of which processes a partition of the input data. The input parameter (memory size) is 8M.

References

[1] Cray assembly language (cal) for cray x1 system reference manual.

[2] John McCalpin. http://www.cs.virginia.edu/stream.

[3] Mingliang Wei, Marc Snir, Josep Torrellas, and R. Brett Tremaine. A near-memory processor for vector, streaming and bit manipulation workloads. Technical Report UIUC DCS-R-2005-2557, 2005.