A Near-Memory Processor for Vector, Streaming and Bit Manipulation Workloads*

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

Technical Report UIUC DCS-R-2005-2557
Dept. of Computer Science, UIUC
February 2005

Abstract

Many important scientific and engineering applications execute sub-optimally on current commodity processors and servers. Specifically, since they frequently use caches ineffectively, they are often heavily bottlenecked by global memory bandwidth. In addition, they sometimes need to perform expensive bit manipulation operations that are not efficiently supported by commodity ISAs. Moreover, an analysis of technology trends suggests that, despite the criticality of some of these applications, future commodity processors and servers are unlikely to be tuned for them.

To address this problem, this paper proposes the design of a simple co-processor on which the main processor can off-load vector, streaming, and bit-manipulation computation. The co-processor is a blocked-multithreaded narrow in-order core with support for vectors, streams, and bit manipulation. It has no caches and a high bandwidth to memory. For this reason, rather than for its actual physical location, we call it *Near-Memory Processor (NMP)*. Our simulations show that a set of scientific applications run much faster on the NMP than on an aggressive conventional processor. Specifically, the speedups obtained reach 18, with a geometric mean of 5.8 for 10 applications.

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

1 Introduction

Many scientific and engineering applications, including several key ones from the defense domain, are not supported efficiently by current commodity processors and servers. These applications often exhibit access patterns that, rather than reusing data, stream over large data structures. As a result, they make poor use of caches and place high-bandwidth demands on the main memory system, which is one of the most expensive components of high-end servers.

In addition, these applications often perform sophisticated bit manipulation operations. For example, bit permutations are used in cryptographic applications [21]. Since commodity processors do not have direct ISA support for these operations, they are performed in software through libraries, which are typically slow.

An analysis of technology trends suggests that future commodity systems are unlikely to be significantly redesigned to support these applications very efficiently. Indeed, high-frequency commodity processors will continue to rely on several levels of caches to minimize main memory bandwidth requirements. Similarly, commodity ISAs are unlikely to include much support for sophisticated bit manipulation operations. For the users of these often-critical applications, the outlook is not very positive.

To solve this problem, this paper proposes to off-load the types of computations that run inefficiently on commodity systems onto a simple co-processor that is especially designed to support them well. Specifically, we propose the design of a co-processor that efficiently supports vector, streaming, and bit-manipulation computation.

Our co-processor is a blocked-multithreaded, narrow in-order core with hardware support for vectors, streams, and bit manipulation. It is closely coupled with the memory controller(s) of the machine. It has no caches, and a high bandwidth to main memory. For this reason, rather than for its actual physical location, we call it *Near-Memory Processor (NMP)*. A key feature of the NMP is the *Scratchpad*, a large local-memory area directly managed by the NMP.

To assess the potential of the NMP, we simulate a state-of-art high-end machine with an NMP in its memory controller. We find that a set of scientific applications run much faster on the NMP than on an aggressive conventional processor. Specifically, the speedups obtained reach 18, with a geometric mean of 5.8 for 10 applications.

This paper is organized as follows: Section 2 provides a brief background on the architectural techniques considered; Section 3 presents the design of the NMP; Section 4 introduces its programming environment; Section 5 evaluates the design; and Section 6 surveys related work.

2 Background

2.1 Vectors, Multithreading, and Streaming

Vectorization is the traditional mechanism used for latency hiding in applications with a regular control flow that fits the data parallel model [20]. Vector loads and stores effect a large number of concurrent memory accesses, typically bypassing the cache. If random memory locations need to be accessed, scatter/gather operations can be used. The vector programming paradigm is well understood and well supported by compilers.

A more general way to hide high memory latencies is to support the execution of multiple threads in the same processor core, so that when one thread stalls waiting for memory, another one can make progress. This approach is called multithreading, and can be implemented in a variety of ways [22]. One very simple implementation involves running a single thread at a time, and only preempting the thread when it encounters a long-latency operation, such as an L2 cache miss or a busy lock. This approach is called blocked multithreading, and was implemented in the Alewife machine [2] and the IBM RS64IV [7]. It has been shown that blocked multithreading can run efficiently with only a few threads or contexts [23].

When multithreading is used, it is very desirable to provide efficient inter-thread communication and synchronization mechanisms between the threads. Producer-consumer primitives are particularly powerful. With these, one can very efficiently support a streaming programming model [8, 13, 14]. A stream program consists of a set of computation kernels that communicate with each other, producing and consuming elements from streams of data. This model suits data intensive applications with regular communication patterns, like many of the applications considered in this paper.

2.2 Single vs Multiple Cores

With a given transistor budget, it is possible to build the NMP as multiple, very simple cores, each with a separate register file and memory area. Alternatively, we can build it as a single, relatively more powerful core with a single register file and memory area. The first approach works well for applications with regular computation flows because it minimizes interference between them. However, it is harder to use with applications that have irregular, data-dependent control flows [4]. In these applications, there may be a large variation in the computation, bandwidth, or footprint requirements of the different control flows. Consequently, a single, more powerful core is more cost-effective. We use this second approach for the NMP.

3 Proposed Architecture

3.1 Rationale

We would like a simple co-processor organization that significantly accelerates vector, streaming and bit manipulation workloads, while being broadly usable for many other classes of applications. For this, we select a blocked-multithreaded core with some architectural support for these three classes of workloads.

The combination of the vector/streaming models and blocked multithreading is attractive, as modest levels of multithreading are sufficient to address the limitations of these models. Specifically, for vector workloads, processor stall caused by short vectors or by highly-variable memory access latencies can often be hidden by preempting the current thread and running another one. Similarly, for streaming workloads, processor stall caused by the imbalance of computation or memory bandwidth between a producer and a consumer stream kernel can usually be hidden by preempting the fast kernel, and running the slow one.

To be able to exploit data locality, the NMP has a large, high-bandwidth, multi-bank local memory area that it directly manages. We call it the *Scratchpad*. The NMP has support for vector load and store operations that move data between memory and scratchpad. Data can be moved from the scratchpad to the vector unit in the NMP using multiple concurrent lanes, thus providing high bandwidth access to vector operands. Vector gather and scatter instructions are also available.

To support streaming efficiently, the NMP supports very low overhead producer-consumer synchronization between concurrent threads. Specifically, each addressable unit in the scratchpad (byte) has a full/empty bit, with logic that can block a thread that attempts to read an empty word or write a full word.

Since the scratchpad is large, it is impractical to save and restore it upon context switch. Thus, the scratchpad is not part of a thread context — the thread context includes only a small number of scalar and control registers. The scratchpad is accessed with virtual addresses and is shared by all the threads running on the NMP. Although using virtual addresses slightly increases scratchpad access time, the overhead is modest if data is processed using long vectors, as address translation is performed only once per vector operand access in the scratchpad. Such virtualization has the added benefit that scratchpad storage associated with threads that are inactive for a long period of time can be lazily paged out (into main memory) and brought back on demand when accessed.

The NMP also includes instructions for bit processing like those of the Cray machines [20]. In particular, it has a bit matrix register that is used for data permuting instructions such as bit matrix multiply. The bit matrix register is also virtualized, so that it does not have to be saved and restored on context switch.

Overall, the resulting architecture is fairly general and can speed-up many classes of applications. However, it is not optimized for cache-friendly applications, and will not be competitive with a conventional superscalar processor for such applications. Thus, we envision its use as a flexible co-processor on which the main processor can off-load computation.

In the following sections, we overview the design (Section 3.2), describe the scratchpad (Section 3.3) and give some details on the ISA (Section 3.4).

3.2 Overview of the Design

Figure 1 shows the NMP in a system like the IBM Power 5. Each memory controller is associated with an NMP. Note that the NMP may be implemented on-chip with the processor, or physically nearer the main DRAM memory.

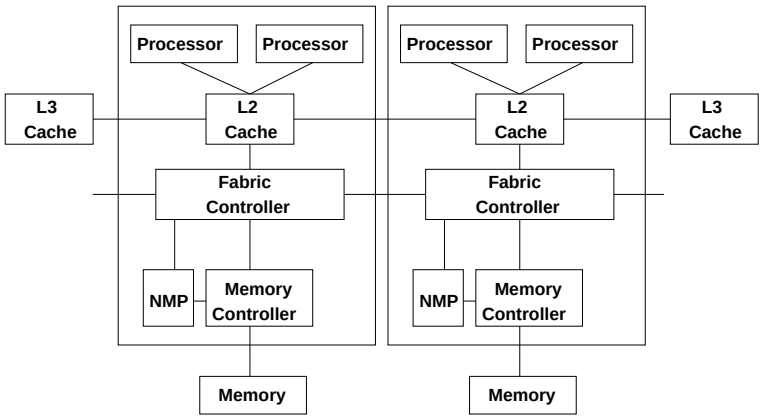

Figure 1: NMPs in a system like the IBM Power 5.

Figure 2 shows the organization of the NMP. The dashed box encloses the NMP. In the figure, the NMP Interface provides an interface for the NMP to communicate with the rest of the system. The main processor(s) communicate with the NMP via the Invocation Register Sets (Section 3.5.1). As soon as a request from a main processor arrives, the Thread Management Unit creates a thread and inserts it into the NMP's job queue.

Figure 3 shows the organization of the NMP core. For simplicity, the core is a low-issue in-order processor. It does not have data caches but, as indicated before, it uses the explicitly-managed fast Scratchpad memory (Section 3.3). It includes scalar functional units, vector/streaming functional units, a set of general-purpose and control registers, and a single Bit Matrix Register (BMR) to permute the bits within a word [1].

To save space, there is only a single BMR. The BMR is tagged with the owner thread ID and is not saved upon context switch. If the hardware detects that a thread is going to access the BMR

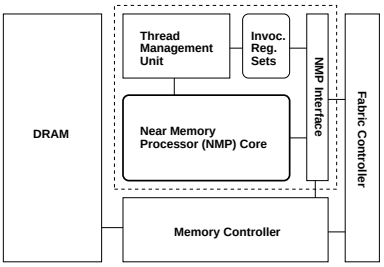

Figure 2: Overall organization of the NMP.

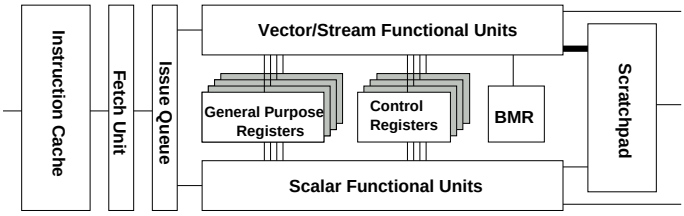

Figure 3: NMP core organization.

with an inconsistent thread ID tag, an exception occurs. The operating system then saves the BMR contents and loads the BMR for the current thread.

All threads running on the NMP share the scratchpad. In addition, they can access any main memory location in the machine. To access memory, an NMP thread uses the same 64 bit virtual addresses as if it ran in the main processor. The NMP has a TLB to cache address translation entries that are kept coherent with other TLBs in the system.

3.3 The Scratchpad

The scratchpad is a general-purpose, explicitly-managed storage area for frequently-accessed scalars, vectors and stream buffers. In addition, it supports fine grain, cross-thread synchronization. Each addressable location (byte) is associated with three one-bit flags: a full/empty bit, an error flag bit, and a mask bit. The first one is for fine-grain synchronization: a synchronized read that consumes the data stalls until the bit is on, while a synchronized write stalls until the bit is off. The error flag bit is used for recording what locations suffered exceptions during the execution of a vector operation. Finally, the mask bit is used to mark the elements of a vector that need to be masked off in a vector operation.

The scratchpad is mapped into the global address space of the system. Consequently, it can be accessed by the main processor(s) using a regular, 64 bit virtual address. Such accesses are snooped to maintain coherence.

Threads running on the NMP address the local scratchpad using a shorter virtual address, currently set at 20 bits. The TLB logic for such accesses is set so that the address translation mechanism never misses when data actually is in the scratchpad. These accesses do not go out on the coherent network and, therefore, are not snooped. Thus, scratchpad locations that are accessed by the NMP should normally not be accessed by other processors or should be accessed in non-cached mode.

An access to the scratchpad may cause a scratchpad page miss, which means that the virtual page containing the requested data is not in the scratchpad. The miss is handled by a special page

miss handler that uses main memory as a paging device, resulting in low page fault overhead.

3.4 ISA Details

The NMP's ISA is targeted to speed-up vector, stream, and bit manipulation operations. In this section, we give some details on the ISA. The full description can be found in [9].

Storage in the scratchpad can be interpreted to hold scalars, vectors, or stream buffers (queues). The interpretation results from the semantics of the instructions used to access the scratchpad. Such instructions specify an opcode, the operands size, the addressing mode, and up to three registers (Figure 4). When direct addressing is used, the register contains a scalar operand. When indirect addressing is used, the register contains a specifier for an operand in the scratchpad. Specifically, it can have a scalar specifier, a vector specifier, or a stream buffer specifier.

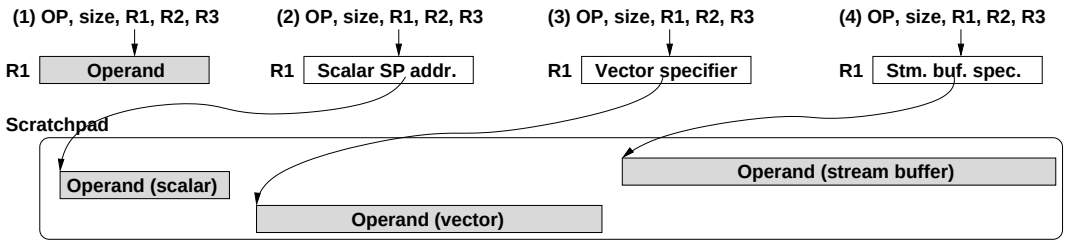

Figure 4: Addressing modes for NMP instructions: (1) direct mode, (2) scalar indirect mode, (3) vector indirect mode and (4) stream indirect mode.

A scalar specifier is a scratchpad address. A vector specifier consists of a vector start address and a vector length. Recall that the size of the vector elements is part of the instruction format. The vector operation automatically uses the mask bits associated with its input operands in the scratchpad storage, and may set the error bits and mask bits associated with its output operand in the scratchpad.

A stream buffer specifier consists of a buffer start address, the buffer length, and a pointer to the head of the buffer (for input operands) or to the tail (for output operands). The size of the buffer elements is part of the instruction format. A stream read can either peek at the element at the head of the buffer, or dequeue it. A stream write deposits an element at the tail of the buffer. These operations update the buffer pointers. The full/empty bits in the scratchpad are used to avoid underflow and overflow. The head and tail pointers wrap around at the boundaries of the buffer.

We require that vectors and buffers fit within one scratchpad page, so that there is a single address translation operation per operand.

Table 1 summarizes the bit manipulation instructions supported by the NMP.

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: $s_{ij} = popcnt(s_k \& bmr_j) \pmod 2$, 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 shifted into the block.
Mix	Bit-interleave higher(lower) half of two words.

Table 1: Bit manipulation instructions.

3.5 Other Issues

3.5.1 Coprocessor Interface

The NMP works coupled with the main processor, using a mode where the main processor is the master and the NMP is the slave. This simplifies the software. However, since the execution time of NMP threads may vary, we need a low-overhead, asynchronous interface between the main processor and the NMP. For this, we use the Invocation Register Sets of Figure 2, which are memory mapped in the main processor's address space.

In our design, the main processor triggers an NMP computation by storing an Invocation Packet into one of the Invocation Register Sets of the NMP. To reduce overhead, this operation is done in user mode, without a system call. The invocation packet includes a pointer to the function to invoke, a pointer to its arguments, and a pointer to a completion flag currently set to zero. The main processor can then regularly poll the completion flag. The NMP signals completion by setting the completion flag. We expect this interface to have very low overhead.

3.5.2 Protection and Virtualization

An NMP may be executing threads on behalf of more than one process running on the main processor(s). These threads need to be protected from each other. Some NMP threads may even belong to processes that are not running in the main processor(s) but are still alive. In order to use NMP resources efficiently, such threads need to be descheduled.

To do so, we manage NMP contexts as memory, piggy-backing on the virtual memory management infrastructure. Specifically, scratchpad space is allocated in swappable pages; each NMP thread is associated with some "low core" scratchpad space that is used to save the thread context. The scratchpad pages are paged to main memory by an external pager when physical scratchpad space needs to be allocated to a newly-invoked thread. The paging mechanism ensures that a thread cannot overwrite scratchpad space or memory used by other threads. However, partial sharing of

the scratchpad space, via stream buffers, is also possible.

3.5.3 Exceptions

Regular virtual memory exceptions caused by vector loads/stores that move data between the memory and the scratchpad are not precise. However, they are restartable and are handled by software. As a result, the NMP is allowed to continue execution from the point where the virtual memory exception occurred. This is the same approach as used in [15].

To simplify the handling of exceptions, individual vector and stream buffers in the scratchpad are always stored sequentially and not allowed to cross pages. Consequently, accesses to one of them can at most cause a single page fault.

Finally, the handling of vector arithmetic exceptions is postponed until the completion of the vector instruction, as in [3]. The faulting elements are marked in the error flag bits of the destination vector.

4 Programming Environment

To use the NMP, we can have at least three different levels of programming complexity and generality. At the simplest level, the NMP can be utilized via a high-level library interface that provides hand-tuned implementations of important scientific or multimedia kernels. A second, more advanced level consists of having libraries separately generated by a compiler. Due to the simple nature of the NMP architecture, it is quite easy to compile the NMP code: the NMP can benefit from mature vectorization technology and from recent work on streaming compilers [10, 16]. Such a compiler, augmented with a run-time library for thread creation, synchronization and termination, can be used to develop new NMP code. At the most ambitious level, one can imagine a compiler that compiles combined main processor/NMP code, choosing what tasks to offload to the NMP.

4.1 The API for the NMP

The NMP has a simple thread-like API. Initially, a main processor calls NMP_Connect(Addr) to establish a connection with an NMP. This is a system call that allocates an Invocation Register Set to the caller process. On success, NMP_Connect() returns a handle N through which the main processor can have user-level communication to an NMP. The NMP selected by NMP_Connect() is the one whose local memory module contains the physical address corresponding to the virtual address *Addr*. From now on, the main processor can use the handle N to spawn threads on the NMP that is "near" to the data at address *Addr*.

The main processor can create multiple threads over time on the NMP without intervention by the operating system. A thread is created by calling Memthread_Create(Function, Arguments, CompletionFlag, N), where *Function* is a pointer to the function to execute, *Arguments* are the

function's arguments, *CompletionFlag* is a simple flag, and *N* is the handle returned from the previous call. This code writes an Invocation Packet to the Invocation Register Set. The NMP indirectly obtains a precompiled thread from the location in memory indicated by the pointer *Function*. This is a user-level invocation. The Memthread_Create() invocation returns a status indicating whether the operation succeeded.

The Memthread_Create() invocation is nonblocking: the main processor resumes execution without waiting for the NMP thread to complete. The main processor may check at a later point if the NMP thread completed by calling Memthread_Wait(CompletionFlag), which blocks the calling thread until the *CompletionFlag* is set. It can also invoke function Memthread_Poll(CompletionFlag) and Memthread_Select(CompletionFlag), whose semantics are the same as poll() and select() in UNIX systems.

Upon completion, the NMP thread sets the completion flag $CompletionFlag$ in memory.

Finally, to disconnect from the NMP, the main processor calls the NMP_Disconnect(N) system call. The parameter *N* is the handle of the NMP to be disconnected.

5 Evaluation

5.1 Evaluation Methodology

To evaluate the NMP concept, we use execution-driven simulations of detailed processor and memory system models. We model the architecture shown in Figure 5, which contains a single main processor and a single NMP. The main processor is a 4-issue out-of-order superscalar with two levels of caches, while the NMP is a 2-issue in-order blocked-multithreaded processor. Main processor, memory controller, and NMP share the same processor chip. The parameters of the architecture modeled are shown in Table 2. Note that the main processor has an aggressive 16-stream hardware stride prefetcher. The prefetcher is similar to the one in [19], with support for 16 streams and non-unit stride. The prefetcher brings data into a buffer that sits between the L2 and main memory.

For the evaluation, we select a number of small applications that we list in Table 3. On average, the applications have 730 lines of C code. The table shows if the applications can be vectorized, use streams, or use bit maniputation instructions. The table also shows the number of concurrent NMP threads used for each application.

The simulated NMP has a MIPS-like ISA, augmented with vector, streaming, and bit manipulation instructions. Since we do not have a compiler that generates vector or stream codes, we hand-coded the vector, streaming, and bit manipulation instructions. These new instructions are captured and simulated by the simulator. All programs are compiled using gcc compiler version 3.2.1.

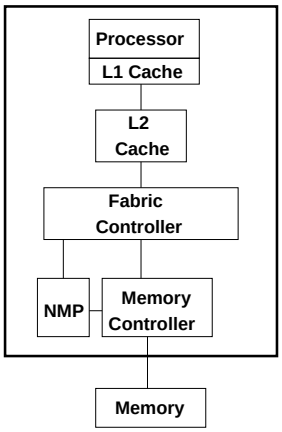

Figure 5: Architecture modeled. The box with the thick boundary is the processor chip.

NMP Parameters	
Parameter	Value
Frequency	4GHz in-order
Issue Width	2
# Scalar FUs	1Int FU, 1FP FU
# Vector FUs	16Int FUs, 16FP FUs
# Lanes	16
# Pending Memory Ops (Ld, St).	128, 128
# Contexts	4
Time to Context Switch	4 cycles

Memory Parameters	
Parameter	Value
L1, L2, Scratchpad size	32KB, 1MB, 64KB
L1, L2 associativity	2-way, 4-way
L1, L2 line size	64B, 64B
Main proc. to L1, L2, memory round-trip latency	2, 10, 500 cycles
NMP to Scratchpad, memory latency	6, 470 cycles

Main Processor Parameters	
Parameter	Value
Frequency	4GHz out-of-order
Fetch Width	8
Issue Width	4
Retire Width	8
ROB size	152
I-window size	80
Int FUs	3
FP FUs	3
Mem FUs	3
Pending Ld/St	16, 16
Branch Pred.	Like Alpha 21464
Branch Penalty	14 cycles
Hardware Prefetcher	16-stream stride
Prefetch Buffer	16KB
Pref. Buf. Hit Delay	8 cycles

Table 2: Parameters of the architecture modeled.

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			4	3DES encryption
PartRadio	X	X		3	Partial radio station
Stream	X			4	Simple vector operations

Table 3: Applications evaluated.

To understand the applications, 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 [17] 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.

5.2 Main Results

Figure 6 shows the speedups of the applications running on the NMP processor over running on the main processor. Recall that the main processor has an aggressive hardware prefetcher (Section 5.1). In the figure, the *Copy*, *Scale*, *Add* and *Triad* bars correspond to the four components of the *Stream* application. The rightmost set of bars are the geometric mean of all the applications.

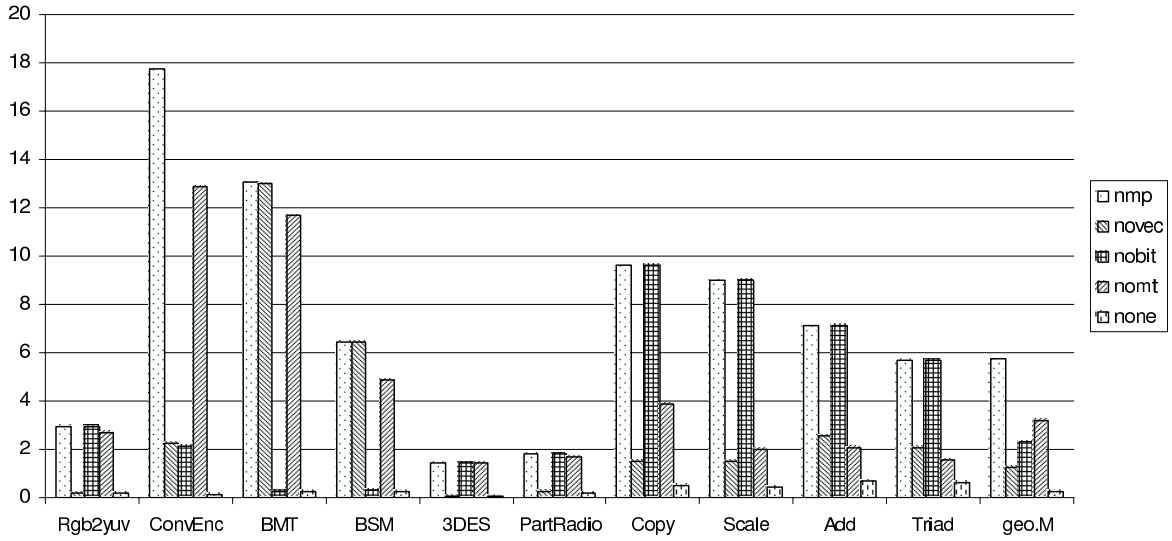

Figure 6: Speedup of the applications running on the NMP over running on the main processor with an aggressive hardware prefetcher. In the figure, *Copy*, *Scale*, *Add* and *Triad* are the four components of the *Stream* application. The rightmost set of bars are the geometric mean of all the applications.

For each application, we show five different bars, to see the impact of the different architectural supports in the NMP. The *nmp* bars correspond to the full fledged NMP architecture. *novec* is the NMP without the vector hardware support. *nobit* is the NMP without the bit-manipulation hardware support. *nomt* is the NMP without the streaming support and running with a single thread. Finally, *none* is the NMP without vector, bit manipulation, and streaming support.

Focusing first on the *nmp* bars, we see that these applications typically run much faster on the NMP than on an aggressive conventional processor with a hardware prefetcher. Specifically, the speedups obtained reach 18, with a geometric mean of 5.8 for the 10 bars. Since the NMP is approximately at the same distance from memory as the main processor (Table 2), the speedups of the NMP do not come from shorter memory latencies. Instead, they come from a better ability to hide the memory latency (and, therefore, reduce stall time) and from architectural support for

several operations common in these applications.

To better understand this effect, Figure 7 breaks down the execution time of the applications into time that the processor is busy executing instructions (*Busy*) and time that it is stalled, mostly waiting on the memory system (*Idle*). The figure shows two bars for each application; the leftmost one is for the execution on the main processor, while the rightmost one is for the execution on the full-fledged NMP. For each application, the bars are normalized to the execution time on the main processor.

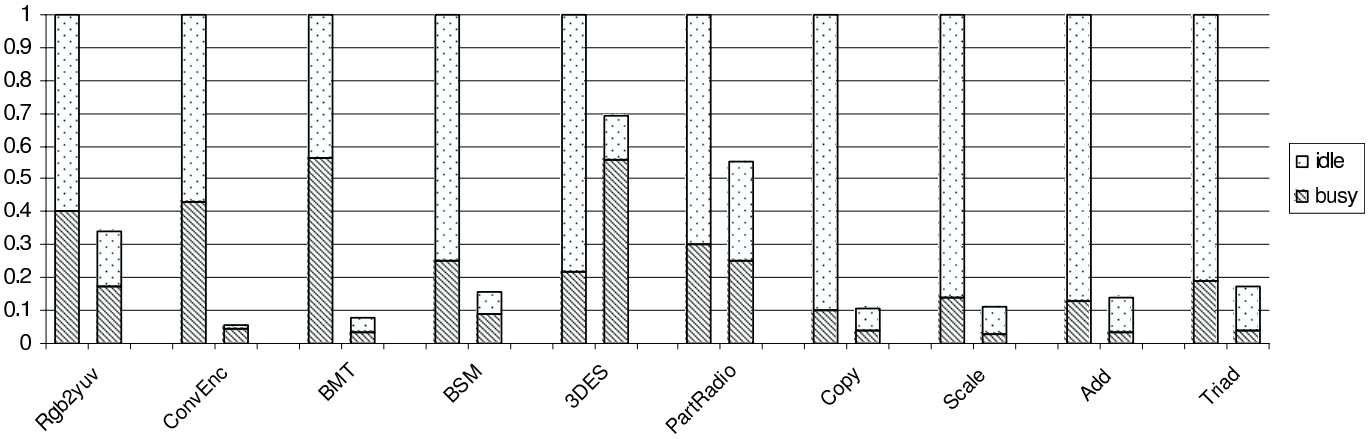

Figure 7: Breakdown of the execution time of the applications on the main processor (leftmost bars) and on the full-fledged NMP (rightmost bars). For each application, the bars are normalized to the execution time on the main processor.

From the figure, we see that most of the execution time reduction of the NMP bars comes from a large reduction in the applications's stall time. This is largely due to the better architectural support in the NMP to hide memory latency. In addition, the busy time also typically goes down in the NMP. This is despite the fact that the NMP is a narrower issue processor, and it should take longer than the main processor to execute the same number of instructions. In reality, the reason why the busy time goes down for the NMP is the better ISA support in the NMP for some of the operations required by these applications. One interesting exception is *3DES*, where the busy time goes up. The reason is that this application does not need the bit manipulation instructions introduced by the NMP.

Going back to Figure 6, we now focus on the *novec* bars. They show that vector support is critical to several of these applications. In particular, *Rgb2yuv*, *3DES*, and *PartRadio* require the vector support in the NMP to deliver any speedup at all.

The *nobit* bars show the importance of the support for bit manipulation. We can see that *BMT* and *BSM* heavily rely on this support. In *ConvEnc*, both vector and bit manipulation support are

necessary to obtain good speedups — if any one is eliminated, the speedup drops substantially.

The *nomt* bars show that the four components of *Stream* (*Copy*, *Scale*, *Add* and *Triad*) need the streaming and multithreading support. If such support or the vector hardware is eliminated, performance drops.

Overall, each of the three supports present in our proposed NMP is important to speed up at least some of the applications considered. Finally, if we eliminate all the three supports (*none* bars), the NMP is much slower than the main processor for all the applications.

6 Related Work

We briefly consider three related areas, namely processing in memory, stream architectures, and multithreaded vector architectures.

6.1 Processing in Memory

Processing in Memory (PIM) or Intelligent Memory architectures integrate logic and DRAM in the same chip. Some of the PIM approaches [11, 12, 18] suggest to replace main memory by PIM chips. Since the in-memory processor directly connects to the memory banks, it has a high bandwidth and low latency to main memory. Results show a significant improvement for a variety of applications. However, PIM chips require significant and likely costly modifications to DRAMs.

Our NMP is different from PIM in that it does not require modifications to the DRAM chips. The NMP can be placed on the processor chip or closer to main memory.

6.2 Stream Architectures

A stream program is organized as streams of data processed by computation kernels. A stream processor is optimized to exploit the locality and concurrency of stream programs. Imagine [14] and Merrimac [8] are two examples of the stream architecture.

The NMP architecture has some of the support of a streaming architecture, but it can be argued it presents a simpler programming interface. The use of blocked multithreading, in particular, avoids the need for explicitly scheduling and multiplexing the kernels on the same processor, facilitates resource (processor and register) allocation, and helps better overcome variance in the execution time of different tasks.

6.3 Multithreaded Vector Architecture

Espasa and Valero [5, 6] showed that multithreading can be applied to a vector processor to greatly improve the resource utilization. In their design, vector registers are part of the context of a thread. Consequently, they are saved and restored when the thread is pre-empted and re-scheduled.

In the NMP, we have explored a different design, where the vector storage is not part of a thread's saved context. Vectors are stored in the scratchpad, which is an area shared by all threads. Not saving the registers in a context switch reduces the overhead.

7 Conclusion

Many important scientific and engineering applications execute sub-optimally on current commodity processors and servers. Such applications would often benefit from architectural support for vectors, streams, or bit manipulation operations. Since commodity processors are unlikely to be modified to support these needs, this paper has proposed a Near-Memory Processor (NMP) co-processor, on which the main processor can off-load these operations.

The NMP is a blocked-multithreaded, narrow in-order core with hardware support for vectors, streams, and bit manipulation. Rather than caches, it has a scratchpad, which is a large directly-managed local-memory area.

To assess the potential of the NMP, we simulate a state-of-art high-end machine with an NMP in its memory controller. We find that a set of scientific applications run much faster on the NMP than on an aggressive conventional processor with a hardware prefetcher. Specifically, the speedups obtained reach 18, with a geometric mean of 5.8 for 10 applications.

References

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

[2] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B-H. Lim, K. Mackenzie, and D. Yeung. The MIT alewife machine: Architecture and performance. In *Proc. of the 22nd Annual Int'l Symp. on Computer Architecture (ISCA'95)*, pages 2–13, 1995.

[3] K. Asanovic. Vector processors. In *Ph.D. thesis, Computer Science Division, University of California at Berkeley*, 1998.

[4] Mattan Erez, Jung Ho Ahn, Ankit Garg, William J. Dally, and Eric Darve. Analysis and Performance Results of a Molecular Modeling Application on Merrimac. In *SC'04*, Pittsburgh, Pennsylvaniva, November 2004.

[5] Roger Espasa and Mateo Valero. Multithreaded vector architectures. In *HPCA '97: Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture (HPCA '97)*, pages 237–249. IEEE Computer Society, 1997.

[6] Roger Espasa and Mateo Valero. Simultaneous multithreaded vector architecture: Merging ILP and DLP for high performance. In *HIPC '97: Proceedings of the Fourth International Conference on High-Performance Computing*, pages 350–357. IEEE Computer Society, 1997.

[7] J.M. Borkenhagen et al. A multithreaded powerpc processor for commercial servers. volume 44, pages 885–894. IBM J. Research and Development, 2000.

[8] William J. Dally et al. Merrimac: Supercomputing with streams. In *SC'03*, Nov. 2003.

[9] Anonymous for review. Omitted for review.

[10] M. Gordon. A stream compiler for communication-exposed architectures. In *MIT Tech. Memo TM-627, Cambridge, MA*, March 2002.

[11] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Jay Brockman, Apoorv Srivastava, William Athas, Vincent Freeh, Jaewook Shin, and Joonseok Park. Mapping irregular applications to diva, a pim-based data-intensive architecture. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 57. ACM Press, 1999.

[12] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an advanced intelligent memory system. In *International Conference on Computer Design*, pages 192–201, October 1999.

[13] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *IEEE Computer*, pages 54–62, aug 2003.

[14] B. Khailany, W. Dally, U. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media Processing with Streams. *IEEE Micro*, 21(2):35–46, March 2001.

[15] Christoforos Kozyrakis. A media-enhanced vector architecture for embedded memory systems. In *Technical Report UCB CSD-99-1059*. University of California at Berkeley, 27, 1999.

[16] P. Mattson, J. Springer, and R. Garrett, C.and Lethin. Streaming and dynamic compilers for high performance embedded computing. In *High Performance Embedded Computing (HPEC) Workshop (6th)*, Sep 2002.

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

[18] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active pages: a computation model for intelligent memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 192–203. IEEE Computer Society, 1998.

[19] S. Palacharla and R. E. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, Apr. 1994.

[20] Richard M. Russell. The cray-1 computer system. *Commun. ACM*, 21(1):63–72, 1978.

[21] Bruce Schneier. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., 1995.

[22] Theo Ungerer, Borut Robič;, and Jurij Šilc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35(1):29–63, 2003.

[23] Wolf-Dietrich Weber and Anoop Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *ISCA*, pages 273–280, 1989.