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

Abstract

Many important applications exhibit poor temporal and spatial locality and perform poorly on current commodity processors, due to high cache miss rates. In addition, they sometimes need to perform expensive bit manipulation operations that are not efficiently supported by commodity instruction sets.

To address this problem, this paper proposes the use of a heterogeneous architecture that couples on one chip a commodity microprocessor together with a coprocessor that is designed to run well applications that have poor locality or that require bit manipulations. The coprocessor supports vector, streaming, and bit-manipulation computation. The coprocessor is a blocked-multithreaded narrow in-order core. It has no caches but has exposed, explicitly addressed fast storage. A common set of primitives supports the use of this storage both for stream buffers and for vector registers.

We simulated this coprocessor using a set of 10 benchmarks and kernels that are representative of the applications we expect it to be used for. These codes run much faster, with speedups of up to 18 over a commodity microprocessor , and with a geometric mean of 5.8.

1. Introduction

Many applications, including several key ones from the defense domain, are not supported efficiently by current commodity processors. 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 systems.

In addition, these applications often perform sophisticated bit manipulation operations. For example, bit permutations are used in cryptographic applications [23]. Since commodity processors do not have direct support for these operations, they are performed in software through libraries, which are typically slow.

Chip densities continue to increase, while our ability to use more gates in order to improve the performance of a single thread seems to have reached its limits; instead microprocessor vendors are moving to multicore chips. While current designs are of symmetric processors, as the number of cores per chip continue to increase, it is reasonable to explore heterogeneous systems with distinct cores that are optimized for different applications. (A recent example of such a design is the CELL processor [10]; due to the limited public information on CELL, we could not compare our design to it.)

The advantage of a heterogeneous design is that one need not modify most of the software, as application and system code can continue running on the commodity core; code with limited parallelism can continue running on a conventional, heavily pipelined core, while code with significant data or stream parallelism can run on the new core. Each of the cores is simpler to design: the design of the new core is not constrained by compatibility requirements and good performance can be achieved with less aggressive pipelining; the design of the commodity core is not burdened by the need to handle wide vectors or other forms of parallelism. Thus, a heterogeneous system may be preferable even if, theoretically, one could design an architecture that combines both.

Three main mechanisms have been used to handle computations with poor locality: vector processing, multithreading and streaming. We show in this paper that these three mechanisms are not interchangeable: all three are needed to achieve good performance. Therefore, we study an architecture that combines all three.

Both streaming and vector processing require a large amount of exposed fast storage – explicitly addressed stream buffers and vector registers, respectively. The two approaches however manage exposed storage differently. We develop an architecture that provides one unified mechanism to manage exposed storage that can be used both for storing vectors and for providing stream buffers.

Streaming and vector provide a model where compilers are responsible for the scheduling of arithmetic units and the management of concurrency. While vector compilation is mature, efficient compilation for streaming architectures is still a research topic; streaming architectures cannot handle well variability in the execution time of

*This work is supported by DARPA Contract NBCHC-02-0056 and NBCH30390004, as part of the PERCS project.

code kernels, due to data dependent execution paths or to variability of communication time in large systems. The problem can be alleviated by using multithreading, where computational resources are scheduled "on demand" by the hardware. We show in this paper how to combine blocked multithreading with streaming and vector processing with low hardware overhead and show that a modest amount of multithreading can be effective to achieve high performance. The NMP also enables a simpler underlying streaming compiler.

Our coprocessor is a blocked-multithreaded, narrow in-order core with hardware support for vectors, streams, and bit manipulation. It is closely coupled with the on chip memory controller. 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 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 use a set of 10 benchmark and kernel codes that are representative of applications we expect to use the NMP for. The focus in this initial evaluation is on multimedia streaming applications, encryption and bit processing. We find that these codes 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.

The main contribution of this paper is in detailing an architecture that integrates vector, streaming and blocked multithreading with common mechanisms that manage exposed on-chip storage to support both vectors and stream buffers. The architecture provides dynamic scheduling of stream kernels via hardware supported fine-grain synchronization and multithreading, which eases a streaming compiler's job. To the best of our knowledge, the design is novel. The evaluation focuses on important benchmarks and kernels. The evaluation shows that all the mechanisms that are integrated in the NMP are necessary to achieve high performance.

This paper is organized as follows: Section 2 briefs the 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.

The results in this paper are preliminary; additional work is needed to fully validate the design.

2. Background

High memory latency is a major performance impediment for many applications in current architectures. In order to hide this latency, one needs to support a large number of concurrent memory accesses, and to reuse data as much as possible once brought from memory.

Vector processing is a traditional mechanism used for latency hiding. Vector loads and stores effect a large number of concurrent memory accesses, possibly bypassing the cache. With scatter/gather, the locations accessed can be at random locations in memory. Vector registers provide the large amount of buffering needed for these many concurrent memory accesses. In addition, vector operations can use efficiently a large number of arithmetic units, while requiring only a small number of instruction issues, a simpler resource allocator, less dependency tracking and a simpler communication pattern from registers to arithmetic units.

The vector programming paradigm is well understood and well supported by compilers. It works well in applications with a regular control flow that fits the data parallel model [22].

A more general method to hide memory latency is to use multithreading, supporting the execution of multiple threads in the same processor core, so that when one thread stalls waiting for memory, another one can make progress [24]. One very simple implementation is the use of *blocked multithreading* that 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 was implemented in the Alewife [3] and the IBM RS64IV [11]. It has been shown that blocked multithreading can run efficiently with only a few threads or contexts [25].

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 [14, 13, 9]. 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.

When the stream model is used, one obtains additional locality by ensuring that data produced by one kernel and consumed by another is not stored back to memory. Stream architectures such as the Merrimac [14] do so by having on-chip addressable stream buffers, and managing the allocation of space in these buffers and the scheduling of producers and consumers in software. The compiler needs to interleave the execution of the various kernels, a task that is not done efficiently by present compilers [15]. Alternatively, one can use blocked mul-

tithreading and suitable hardware supported synchronization to ensure that the producer is automatically descheduled and the consumer is scheduled when data has been produced and is ready to be consumed. This leads to a simpler target model for compilers, as they compile sequential threads and synchronization operations between threads. This design also handles better tasks with nondeterministic execution time.

3. Proposed Architecture

3.1. Rationale

As discussed in the previous section, vector processing is a well understood, easy to implement mechanism for hiding memory latency, with good software support. Streaming architectures provide a more general latency hiding mechanism, at the expense of a more complex programming model, more complex hardware and the requirement for more advanced compiler technology. However, the streaming model fits well streaming applications where a sequence of kernels are pipelined. Support for the streaming model can be simplified if one uses multithreading, since software does not need to handle the interleaved execution of multiple kernels. Multithreading also handles kernels with variable execution time better.

It turns out that a set of common mechanisms can be used to exploit on chip storage both for vector registers and for stream buffers. With such addressable common storage, it is possible to keep a relatively small amount of state for each executing thread, so that context switching is not expensive; blocked multithreading can be implemented at a relatively low cost and can be used efficiently. Thus, we choose to implement the NMP as an engine that combines vector processing, streaming and blocked multithreading. Finally, we added bit manipulation logic to support bit-oriented applications. As it turns out, all these mechanisms are needed to achieve performance on the applications we consider.

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 stalls 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 stalls 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.

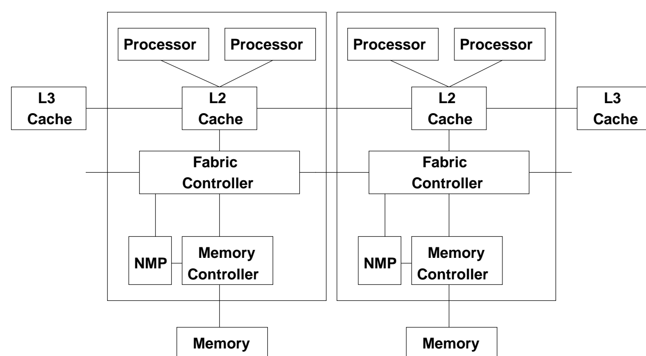

Figure 1: NMPs in a system like the IBM Power 5.

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*. To support streaming efficiently, the NMP supports very low overhead producer-consumer synchronization between concurrent threads, using full/empty bits in the scratchpad. The design is similar to the one used by the HEP [6] and MTA machines[4].

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. Since threads running on the same NMP can belong to distinct processes, we need to provide address protection in the scratchpad. We do so by using virtual addressing. 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 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 [22, 1]. In particular, it has a 64×64 bit matrix register that is used to compute a 64 bit boolean vector-matrix product. Such a product can be used, in particular, to compute an arbitrary 64 bit vector permutation, transpose a 64×64 bit matrix, etc. 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. Our initial evaluation is focused on vector, streaming and bit manipulation applications, as these are most challenging for a conventional processor.

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

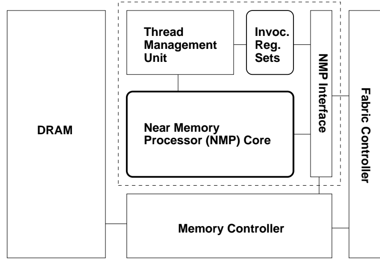

Figure 2: Overall organization of the NMP.

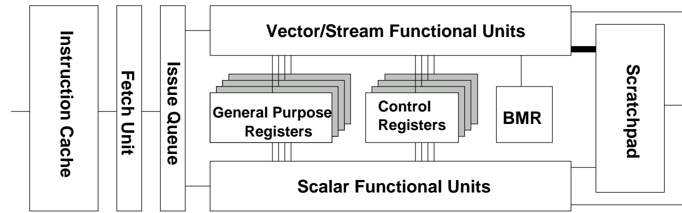

Figure 3: NMP core organization.

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 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 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 runned in the main processor. Accesses of the NMP to the main memory are handled the same way as accesses by the main processor are handled: they are broadcast on the coherence fabric and snooped by caches in the system. 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 an explicitly-managed storage area for frequently-accessed scalars, vectors and stream buffers. The vectors and streams in the scratchpad are stored sequentially. Data can be moved between memory and scratchpad using vector load and store instructions, including strided access and scatter/gather. The vector units process vectors that are contiguous in the screatchpad. One can use masks to selectively perform operations over elements in a vector. (This is similar to the model provided by a vector register). Thus, one can implement the scratchpad using a multi-banked memory with separate lanes from the banks to the vector units, and a barrel shifter to align vectors.

The NMP supports fine grain, cross-thread synchronization. Each addressable location (byte) is associated with three one-bit flags: a full/empty bit[6], 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. (Vector architectures store the mask bits in separate registers, so that the same data can be controlled by different mask vectors; we have not found the need for this extra flexibility in the kernels that we have studied so far.)

For reasons explained in section 3.1, the scratchpad is accessed using virtual addresses: the storage is divided into pages (these need be of the same size as main storage pages). Threads running on the NMP address the local scratchpad using a short virtual address, currently set at 20 bits (8 bit page number and 12 bit displacement). The NMP has a TLB that holds entries for all the pages present in the scratchpad. A TLB miss causes an exception that blocks a thread and is handled by a main processor.

Threads also access the main memory using regular (64 bit) addresses. TLB entries are also required for main memory addresses. We can use a common TLB or two separate TLBs.

Accesses to the main memory are snooped by the caches of the regular processors, hence are coherent. No snooping occurs when the NMP accesses the local scratchpad.

The main processors can access the scratchpad data (including the additional bit flags), but these accesses are not coherent, and the mapping (e.g., of the extra bits) is not straighforward, so that these accesses normally occur only in supervisory mode (e.g., for paging a scratchpad page to memory). The normal mode of operation is that the NMP pulls data from memory (or caches) to the scratchpad and pushes it back.

3.4. Instruction Set

In this section, we give some details on the NMP instruction set. The full description can be found in [26].

NMP instructions are 32 bits. For our simulations, we use an augmented MIPS instruction set [21]. New addressing modes are added to handle streams and vectors.

Storage in the scratchpad can be interpreted to hold scalars, vectors, or stream buffers, i.e., circular buffers holding queues. The interpretation results from the semantics of the instructions used to access the scratchpad and from information stored in registers.

Instructions specify an opcode, the operands size (byte, half-word, word, etc.), the addressing mode (Figure 4), and up to three registers. 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. Bits are included in the register to distinguish different specifiers.

Thus, an instruction `ADD size mode R1 R2 R3` will add two operands specified by `R1` and `R2` and will store the result in a location specified by `R3`. `size` specifies whether the additions are performed on bytes, half-words, words or double-words; `mode` specifies whether each operand is a scalar contained in the specified register (direct addressing) or a scalar, vector or streaming buffer stored in the scratchpad (indirect addressing); not all possible combinations are supported.

A few extra bits are needed in the opcodes to encode operand size (currently 5 choices) and the mode (2 choices, direct or indirect). The extra bits required for these fields are obtained by sacrificing some bits from existing fields, e.g., the shift amount and immediate field.

A scalar specifier is a scratchpad address. The specifier may also specify that the access is conditional on the full/empty bit value in the scratchpad (see below). This can be used for thread synchronization.

A vector specifier consists of a vector start address and a vector length (number of operands).

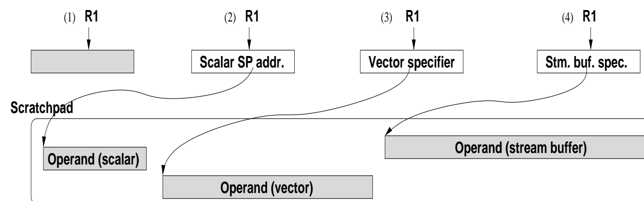

Figure 4: Addressing modes for NMP instructions: (1) direct mode, (2) scalar indirect mode, (3) vector indirect mode and (4) stream indirect mode.

A stream buffer specifier consists of a buffer start address, the buffer length, number of elements to operate in one operation and a pointer to the head of the buffer (for input operands) or to the tail (for output operands). An input operand is dequeued from the head of the buffer and the head pointer in the register is updated; the thread blocks if the queue is empty. An output operand is enqueued at the tail of the buffer and the tail pointer in the register is updated; the thread blocks if the queue is full. The pointers wrap around at the boundaries of the buffer.

All the specifiers fit in a 64 bit register (remember that addresses have 20 bits).

The three operands of an instructions can be all scalars (from a register, or from the scratchpad accessed via a scalar specifier); they can all be vectors of the same length (accessed via a vector or stream buffer specifier). One can also mix scalars and vectors as input operands, in which case the scalar is expanded to the vector length.

If the operands are vectors then the operation 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.

The full/empty bits in the scratchpad are used to avoid underflow and overflow in stream buffers: a consumer marks the element as empty and a producer marks the element as full. Note that the head (resp. tail) of the queue is stored in a register of the consumer (resp. producer), and is not shared; the full/empty bits are in the scratchpad and are shared (a stream buffer is stored in a page that is accessible both by producer and consumer). The current design does not directly support multiple producers or multiple consumers; an additional multiplexing thread is needed to support such. This limitation has not proven a problem with the kernels considered so far.

The scrachpad supports six access types: load, load-if-full, load-if-full-and-mark-empty, store, store-if-empty and store-if-empty-and-mark- full. These access types can be specified explicitly by a scalar specifier; buffer specifiers implicitly require the use of load-if- full-and-mark-empty (for inputs) or store-if-empty-and-mark-full (for outputs). The logic to update the head or the tail of a stream buffers, use mask bits or set error bits is in the functional units.

New instructions are added to move data between memory and scratchpad using strided or indirect vector loads and stores (scatter/gather), as these require more than one register argument. If the destination is a stream buffer, the load becomes a stream load. Also, new instructions are added for bit manipulation; these are summarized in Table 1. The *Sshift* instruction can be used to shift vectors.

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.
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.

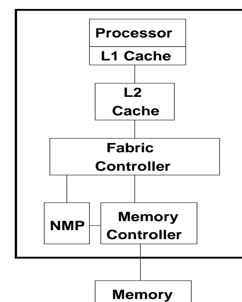

Figure 5: Architecture modeled. The box with the thick boundary is the processor chip.

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. The main processor triggers an NMP computation by storing an Invocation Packet into one of the Invocation Register Sets of of Figure 2, which are memory mapped in the main processor's address space. The mapping into user space is done in superviser mode, while the storing Invocation Packet operation is done in user mode, without a system call. The packet is moved immediatly into a queue, clearing the register for a new invocation from the same process(an exception occurs if the queue is full). The invocation packet includes a pointer to the function to invoke, a pointer to its arguments (including a pointer to a completion flag currently initialized 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, piggybacking 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 and Context Switching

A thread may get blocked in the middle of executing a vector computation. The NMP is designed to be able to continue a vector operation from the point where it was stopped. This is the same approach as used in [17]. The same logic is used to handle virtual memory exceptions that happen during the execution of vector loads/stores that move data between memory and the scratchpad.

The handling of vector arithmetic exceptions is postponed until the completion of the vector instruction [5]. The faulting elements are marked in the error flag bits of the destination vector.

4. Programming Model

4.1. Processor-NMP Communication

Threads are created by processes running on a main processor using a system call that returns a handle – effectively a pointer to an Invocation Register Set; the call fails if no Invocation Register Set is available. Another system call can be used to kill the thread and free the handle.

The communication model between processor and coprocessor is that of an asynchronous procedure call: code running on the main processor can invoke a function on the coprocessor; the invocation specifies the thread to run this function, a pointer to the function, and arguments. Normally, one of the argument will be a location of a flag to be set by the invoked function upon completion. Thus, the main processor can poll for invocation completion or block until completion.

4.2. Thread Scheduling

A thread executes only one function at a time, picking from its queue a new invocation to execute when the previous one has completed. The NMP hardware schedules runnable threads round-robin. A running thread executes until it exits, blocks on a synchronization, or idles on a high latency memory access, at which point it is descheduled.

The hardware does not prevent livelock or deadlock; this is the programmer's responsibility. The hardware however maintains sufficient state so as to allow livelock or

deadlock detection by the system, e.g., the time of the last execution by a thread and the last instruction executed.

4.3. Compilation and Run-Time

Our current programming model uses library calls for thread creation, thread termination and thread synchronization and a compiler to generate thread code. The run-time supports the allocation and deallocation of thread structures and scratchpad space, while thread synchronization is directly supported by hardware. We have not yet developed a compiler for the NMP new instructions and addressing mode; currently, we insert additional instructions and vector code manually in the source code. The compilation of thread code from high level language requires added support for vectorization and for the generation of bit manipulation instructions; this does not require new compiler technology as such capability has been available in commercial compilers for a long time.

5. Evaluation

5.1. Evaluation Methodology

To evaluate the NMP concept, we use an execution-driven simulator [2] with a detailed model of the main processor, the coprocessor and the memory system. 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 [20], 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 manipulation instructions. The table also shows the number of concurrent NMP threads used for each application.

The simulated NMP has a MIPS-like instruction set, 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. Details of the applications can be found in [26].

5.2. Main Results

Figure 6 shows the speedups of the applications running on the NMP over the same applications 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 [18]. 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 running with a single thread. Finally, *none* is the NMP without vector, bit manipulation, and streaming support. We did not try to run without streaming support and with all other features on: with no streaming support, threads would need to communicate and synchronize through memory, resulting in very poor performance.

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.

From the figure, we see that most of the execution time reduction of the NMP bars comes from a large reduction in the application's stall time. This is largely due to the better architectural support in the NMP to hide memory latency. The support includes both vector instructions with long vectors and blocked multithreading. This is consistent with the work of Espasa and Valero [7] that has shown that multithreading is necessary (in addition to decoupling) to improve the resource utilization of vector processors.

In addition, the busy time also typically goes down in

NMP Parameters	
Parameter	Value
Frequency	4GHz in-order
Issue Width	2
# Scalar FUs	1Int FU, 1FP FU
# Vector FUs	1Int FU, 1FP FU
# Lanes	16
# Pending Memory Ops (Ld, St).	128, 128
# Contexts	4
Time to Context Switch	4 cycles
Policy for Context Switch	Switch after 20 idle 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
Bandwidth b.t. vec. units and Scratchpad, Scratchpad and memory	256GB/s, 32GB/s

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

Table 3: Applications evaluated.

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 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 is eliminated, performance drops due to the limited number of in-flight memory operations (short load/store queue).

Overall, each of the three supports presented 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. It is because the NMP has a low-issue in-order core.

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 [12, 16, 19] 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 merged DRAM logic that has high production cost.

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 [9] and Merrimac [14] are two examples of the stream architecture.

Impulse[27] expands the traditional memory hierarchy by adding address translation hardware to the memory controller. Data items whose physical DRAM addresses are not contiguous can be mapped to contiguous shadow addresses, which are the unused physical addresses. The memory controller can compact sparse data into dense cache lines and feed the processor with a stream of data.

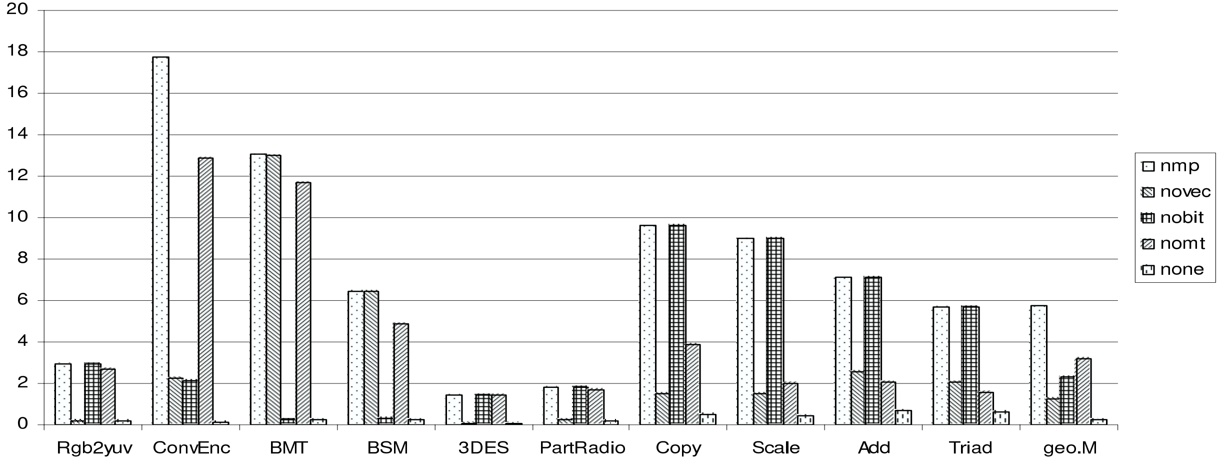

Figure 6: Speedup of the applications running on the NMP over running on the main processor with an aggressive hardware prefetcher. *Copy*, *Scale*, *Add* and *Triad* are the four components of the *Stream* application. The rightmost set of bars are the geometric mean of all the applications.

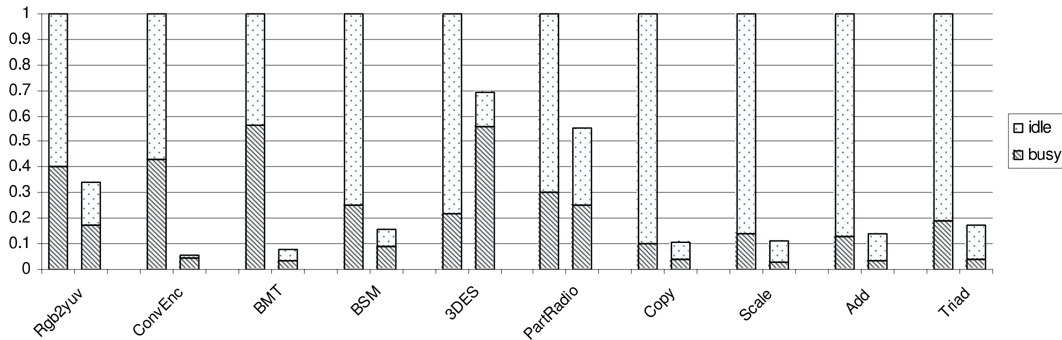

Figure 7: Breakdown of the execution time of the applications on the main processor (leftmost bars) and on the full-fledged NMP (rightmost bars). For each application, the bars are normalized to the execution time on the main processor.

The NMP architecture has some of the support of a streaming architecture, but it can be argued it enables a simpler streaming compiler. 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. The NMP can perform functions similar to the ones implemented in Impulse.

6.3. Multithreaded Vector Architecture

Espasa and Valero [7, 8] 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 preempted and rescheduled.

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

We proposed in this paper a design for an engine that can support efficiently both vector and streaming applications, while providing a simpler interface than a streaming engine where all instruction scheduling is under software control. We believe this combination to be novel. We showed that this engine supports efficiently vector benchmarks, streaming benchmarks and applications requiring bit manipulations. While not demonstrated explicitly in the paper, it is also the case that the streaming compilers for the NMP would be simpler. There was no need for a sophisticated compiler to fuse the kernels.

We expect that increases in chip density will lead to the development of heterogeneous architectures, where functions now provided by external engines, such as GPUs, will be be integrated on chip. CELL is an early example of this trend. Our work shows the potential performance advantage of such an approach in an important domain. The initial evaluation presented in this paper indicates that a chip that contains an NMP in addition to a regular processor can perform significantly better than a regular processor on its own. Of course, future chips could contain multiple NMPs and multiple commodity

processors. While we did not compare explicitly to a commodity processor augmented with a SIMD unit, we believe that the comparison would not be very different since the main performance bottleneck is the exposed memory latency, not the ALU speed.

The initial evaluation used simple kernels because of the lack of a compiler and the need to hand code vector and streaming operations. We hope to follow up in the future with evaluations of more complete applications, including applications and kernels commonly used in high performance scientific computing. This would require a more developed software environment. Also, we hope to study additional variations in the NMP design, including systems with a smaller number of vector lanes.

References

[1] Cray assembly language (CAL) for Cray X1 system reference manual.

[2] http://sesc.sourceforge.net/.

[3] 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.

[4] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 4th International Conference on Supercomputing*, pages 1–6. ACM Press, 1990.

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

[6] B.J.Smith. Architecture and applications of the HEP multiprocessor computer system. In *Society of Photo-optical Instrumentation Engineers*, pages 298: 241–248, 1981.

[7] 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.

[8] 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.

[9] B. Khailany et. al. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, March 2001.

[10] D. Pham et al. The design and implementation of a first-generation CELL processor. In *ISCC 2005: Proceedings of IEEE International Solid-state Circuits Conference*, 2005.

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

[12] Mary Hall et al. 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.

[13] Ujval J. Kapasi et. al. Programmable stream processors. *IEEE Computer*, pages 54–62, aug 2003.

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

[15] Pat Hanrahan. private communication.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] Charles Price. MIPS IV instruction set. 1995.

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

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

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

[25] 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.

[26] Mingliang Wei, Marc Snir, Josep Torrellas, and R. Brett Tremaine. A brief description of the NMP ISA and benchmarks. Technical Report UIUC DCS-R-2005-2633, 2005.

[27] L. Zhang, Z. Fang, M. Parker, B. Mathew, L. Schaelicke, J. Carter, W. Hsieh, and S. McKee. The impulse memory controller, 2001.