SAND2007-0896C

An Architecture to Perform NIC Based MPI Matching

Abstract

Modern supercomputers aggregate thousands of microprocessors through a high performance network. The dominant programming model for these massively parallel systems is the Message Passing Interface (MPI). To improve performance, most of these systems place a processor on the network interface controller (NIC) that handles some fraction of the MPI processing. Unfortunately, this processing inherently involves traversing a linked list and invoking a matching function for each list item. This is a task that microprocessors perform extremely poorly, but that is critical to the performance of the system. Furthermore, the traditional network processor approaches of multicore and multithreading map poorly to the problem because the list is a shared data structure. While this processing is simple enough to be implemented directly in hardware, a pure hardware implementation can be extremely inflexible and lead to extremely high risk. This paper presents a novel, programmable architecture for a processor to handle the matching function. The matching engine approaches the performance of a direct hardware implementation while maintaining a high degree of flexibility and programmability. More importantly, it requires a dramatically smaller area than a conventional processor while achieving significantly higher performance.

1 Introduction

By far, the dominant programming model for modern supercomputers leverages the Message Passing Interface (MPI)[9, 16]. The most commonly used functions for data transfer are the blocking and nonblocking variants of two-sided, point-to-point transfers. To transfer data, one processor initiates an `MPI_Send` while the receiving processor initiates an `MPI_Recv` with corresponding *MPI envelope* information. These two-sided operations require *MPI matching* at the receiver to resolve incoming messages to matching receives. In the nonblocking variants (`MPI_Isend` and `MPI_Irecv`), the sender and receiver can both post a series of nonblocking operations. At the receiver, this translates into a linked list of *posted receives* that must be traversed each time a message is received. The actual matching operation is one of the more computationally complex steps of this traversal and has parallelism that is only bounded by the bandwidth to load the list item into the

processing core. Because matching can be decoupled from the latency dominated list traversal operation, this paper focuses on an architecture to do the matching operation quickly.

Traditionally speaking, network interface controllers (NICs) have included an embedded microprocessor to offload matching operations. Common examples of this include Myricom[18], Quadrics[19], and Cray[2] products; however, it has been observed that this approach can lead to significant increases in message latency under some realistic usage scenarios[23]. Given that the match time *per item* is over $3\times$ the memory access latency (and matching exhibits spatial locality that exploits the cache), this points to limited parallelism within the processor as a significant factor for match time.

To address the issue, a dedicated hardware solution has been proposed for traversing these lists and performing the matching operation[26]. While the MPI matching operation can certainly be implemented entirely in hardware, practical considerations make it undesirable to do so. Most systems evolve the lowest level network API, the implementation of that API, and even aspects of the MPI header format over the lifetime of the system from concept through end of life. Thus, it is desirable to have a more general purpose design that is *programmable*, but at the same time the required flexibility is limited and the design can be heavily customized for the MPI matching problem.

We propose a microcoded engine to process MPI list item matches. It is structured with two ALUs fed by two independent register files with the ability to pass data between the ALUs. Both ALUs are capable of operating in a SIMD manner at 2 byte boundaries within an 8 byte word; however, the two ALUs support different types of operations. One supports typical binary operators, while the other is designed to efficiently implement ternary matching to deal with wildcarded matching entries. Altogether, the microcoded engine approaches the performance limit of the bandwidth to local memory for matching operations.

To evaluate this microcoded engine, we compare it to an embedded microprocessor design point (comparable to current practice) and a multithreaded design point (comparable to what is typically used in network processors). We found that the microcoded engine achieved 94% of the performance of a comparable hardware unit (as limited by the local memory bandwidth) when only 10 list items are traversed, and the embedded microprocessor achieved only 34% of this potential, despite having twice the memory bandwidth. Similarly, a 16 core multithreaded design point only achieves 52% of this potential, despite having $4\times$ the memory bandwidth. The remarkable observation here is that through architectural specialization, it is possible to achieve hardware levels of performance in a programmable processor without the area overhead of a conventional processing approach. An *extremely* conservative estimate places the microcoded match unit at $4.6\times$ smaller than the embedded processor and $3.8\times$ smaller than a *single* multithreaded core.

In the next section, we present related work. In Section 3, we place the work in context by presenting an overview of the matching problem followed by a brief overview of the network interface architecture. Details of the proposed microcoded architecture are then presented in Section 4. Our methodology is explained in Section 5 followed by results in Section 6. Finally, we present conclusions in Section 7 and future work in Section 8.

2 Related Work

Relatively little work has been devoted to the problem of MPI matching. While Quadrics has used a customized processor to perform matching on the network interface for many generations[19], the newest hardware (the Elan5) simply increases the number of thread units rather than specializing the processors. Notably, these processors must implement general code; thus, they cannot be particularly specialized to the matching problem. Similarly, the network interface for the Cray XT3 machine [2] implements the Portals [4] programming interface using a truly general purpose PowerPC 440 embedded processor. However, the embedded processor is ill-suited to quickly traversing the posted receive queue and must also share time with other tasks.

To address potentially long linked lists, research has considered reducing the search cost by using hash tables [18, 21]. However, while a hash table can significantly reduce the time needed to find a matching entry, it also increases the time needed to insert an entry into the list. Because of the high turn over rate inherent in MPI processing, the increase in insertion time is prohibitive. The hashing process is also complicated by the need to support wildcard matching and maintain ordering semantics; thus, the approach has largely been abandoned. There is also a significant amount of previous work on using the general processor on the network interface to implement other operations (MPI collectives, for example) efficiently [5, 6, 17]. Similarly, these approaches focus on protocol optimizations and efficient data movement operations rather than list traversal.

On the surface, MPI matching appears closely related to the much more broadly studied field of network intrusion detection (NID). Network intrusion detection works by matching the contents of network packets against a list of signatures for known exploits, whereas MPI matching must match the MPI envelope information in a packet against the list of posted receives. Work on network intrusion detection includes work on algorithms that operate well on network processors [8, 15] to work on hardware accelerators running in FPGAs [22]. Both of these approaches work by allowing parallel searches through the signature database. Although MPI matching is very similar to the string matching done in NID, there are two main differences that prevent these approaches from working with MPI matching. First, the NID signature database is essentially static (at least for long periods of time) and, thus, leverages off-line processing to make the matching operate quickly. In MPI matching, however, there is high list turnover, making it prohibitively expensive to use off-line types of calculations. Second, MPI matching must maintain strict ordering semantics, whereas NID generally does not.

Another common matching computation in network processing is longest prefix matching, where a router must determine where to route a packet based on the routing table. Longest prefix matching attempts to match the destination with the most complete (i.e. having the fewest number of wildcards) routing rule. Current work has studied using network processors backed with ternary content addressable memories (TCAM) to accelerate this matching[1]. Unfortunately, the MPI matching problem cannot be formulated as a longest prefix match due to the ordering constraints. Also, while a TCAM can prioritize based on longest prefix, it is otherwise inherently unordered. The TCAM approach can be adapted to support MPI matching

by adding ordering into the ternary structure[24]. While this method works well for a small number of entries in the posted receive queue, longer queue lengths still require a linear traversal of those items not in the TCAM structure.

3 System Context

The implementation of the matching operation fits into an overall system context that is defined by the MPI matching problem, as described briefly below. Solving the matching problem, however, requires a specific instantiation on a network interface. Our basic assumptions about the network interface are described following the discussion of MPI matching.

3.1 The MPI Matching Problem

When offloading MPI processing to a network interface, MPI matching is typically abstracted into a lower level network API. For this work, we will consider the Portals API[3, 4], since it is an open specification that abstracts the MPI matching functionality. The (relatively verbose) matching code required by Portals is shown in Figure 1.

A few general comments about the code in Figure 1 are in order. First, the outer loop of the code traverses a linked list that makes up the equivalent of the MPI posted receive queue. At each position, there is an `memd` structure containing a *match entry* and *memory descriptor* which is 64 bytes long and has numerous subfields ranging from 16 bits (e.g. process ID) to 64 bits (e.g. address, match bits). A corresponding, but slightly smaller, header arrives on the incoming message. Second, many of the fields have the ability to wildcard the field (e.g. `match_id_pid == ANY`) or individual bits (e.g. `me->dont_ignore_bits`). Third, the range of operations includes all variants of compares, ternary operations ("don't care bit" masking), and basic arithmetic operations. Finally, virtually all of the comparisons are effectively parallel and offer the potential for numerous concurrent operations — if the architecture can support it. More importantly, that concurrency is free if the list item is going to be retrieved anyway.

In terms of memory access properties, the code has interesting characteristics in terms of both memory latency and memory bandwidth. The linked list traversal certainly incurs the memory latency hit for each list item *if the list is not in cache*. In contrast, the header is in cache after traversing the first list item. At the same time, in the most common case, the code short-circuits after the first test if the match will fail; thus, a relatively small part of the cache line that is loaded is actually used. Although the loop is designed to short-circuit on a failure, with the tests prioritized based on the most common failure conditions, full matching can fail at any point along the path. Thus, any matching unit must be designed to handle all cases.

3.2 Network Interface Context

When the MPI matching problem is solved on the network interface, it is exclusively a receive side problem. It requires inspecting the incoming message headers to determine where the data should be placed and informing a DMA engine. Thus,

```
struct memd*
match(struct memd* memd, const struct ptlhdr * hdr, uint32_t src_nid,/* IN */
      uint64_t* offset, uint32_t* mlength /* OUT */) {

  /* Loop through ME list, looking for a match */
  struct memd* current;
  for ( current = memd; current != NULL ; current = current->next ) {
      struct user_me *me = &current->me; struct user_md *md = &current->md;

      /* Check the match bits */
      if ((me->match_bits ^ hdr->match) & me->dont_ignore_bits) continue;

      /* Check the match id NID and PID */
      if (me->match_id_nid != ANY && me->match_id_nid != src_nid) continue;
      if (me->match_id_pid != ANY && me->match_id_pid != hdr->initiator_pid) continue;

      /* The MD must be valid and must be active (non-zero threshold)*/
      if (!md_valid(current) || !md->threshold) continue;

      /* MD must be configured to respond to the incoming operation type */
      if (hdr->op == PTL_MSG_PUT && !respond_to_puts(md)) continue;
      else if (hdr->op == PTL_MSG_GET && ! respond_to_gets(md)) continue;

      /* Unpack the MD's length */
      uint64_t md_len = get_length(md);

      /* Calculate the effective offset */
      if (use_remote_offset(md)) {
        *offset = get_remote_offset(hdr);
        if (*offset >= md_len) *offset = md_len; /* clamp so math below works */
      } else {
        *offset = get_implicit_offset(current);
      }

      /* Calculate the amount of space remaining in the MD */
      uint64_t remain = md_len - *offset;

      /* MD is inactive if use_max_size() and remain is less than max_size */
      if (use_max_size(md) && !allow_truncation(md)) {
        uint64_t max_size = get_max_size(md);
        if (remain < max_size) continue;
      }

      /* Determine the length to receive into the MD */
      if (hdr->length <= remain) *mlength = hdr->length;
      else if (allow_truncation(md)) *mlength = remain;
      else continue;

      /* Set the offset output and return the match*/
      return current;
    }
  return NULL;       /* no match was found */
}
```

Figure 1. Match function code

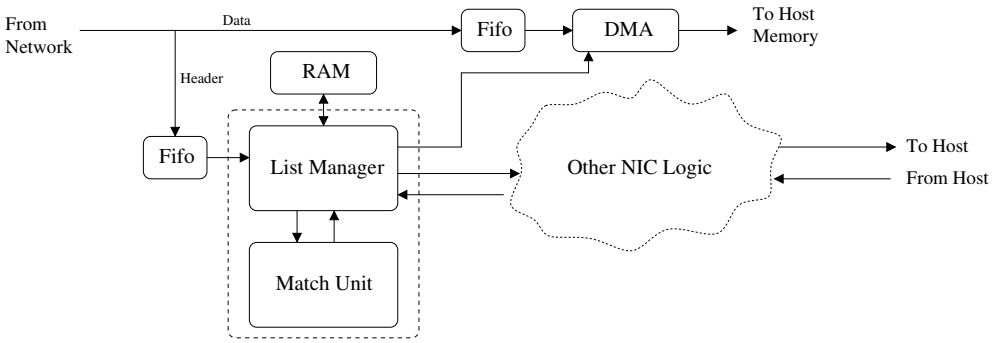

Figure 2. Block diagram of the NIC receive side architecture.

a construct is needed to traverse the posted receive queue and find a matching entry. Figure 2 shows the example of a receive side NIC architecture used for this work. The dashed box designates the functionality that would typically be served by an embedded microprocessor. It interacts with a FIFO structure to deliver network headers and a DMA to place the data into memory.

The alternative presented in Figure 2 replaces the embedded microprocessor with a *list manager* and *match unit*. The list manager provides support for adding items to or deleting items from the posted receive queue. It is also responsible for streaming list items to the match unit for matching, and for providing information to the host and DMA engine about a matching receive. The match unit's sole responsibility is to compare an incoming header with the items in the posted receive queue to see if there is a match.

The list manager manages a small cache of list items (two or three 64 byte items) to cover the round-trip time to the local memory. This essentially hides memory access time, allowing header processing to proceed without stalling. When a new header is received, the list manager pulls it out of the header buffer and passes it to the match unit for processing. At the same time, it starts memory requests for the first list item not in the cache. Immediately after the header is sent to the match unit, the list manager starts streaming in the list items, starting with those in the cache. The list manager receives either a "Match Failed" or a "Match Successful" for each list item sent to be matched. When confirmation of a successful match is received, the list manager completes sending of the current list item, then sends an "end list" command. On a successful match, the match unit also sends an offset and a length for the destination of the message in the target buffer. This information is used by the list manager to create the appropriate DMA commands for sending the message data to the host's memory. The list manager also sends an event, either directly or through another NIC component, to the the host letting it know about the received message.

4 Match Unit Architecture

In many ways, the match unit is a general purpose processing engine; however, the match unit has been specialized in multiple ways. Foremost, inputs and outputs arrive through FIFO constructs, rather than from user specified memory locations. FIFOs provide a simple interface mechanism with other components in the system — namely the list manager. Inputs arrive into two independent register files that feed independent ALU and ternary ALU datapaths. Results from these operations are aggregated through a predicate register file with a predicate combining unit to affect branch behavior. Overall, on every cycle, the match unit microarchitecture can: 1) input an item, 2) output or copy an item, 3) perform an ALU operation, 4) perform a ternary operation, 5) perform a predicate merge operation, and 6) resolve a branch.

As a by-product of the extensive concurrency available, the highest level of programming for the match unit uses assembly language, because the level of and specificity of concurrency within the core would be difficult to express efficiently in a high level language. The assembly language is translated directly into microcode, and the characteristics of the matching code written for the match unit are discussed in Section 4.3.

4.1 Motivating Objectives

The architecture of the match unit was driven by three main considerations: high throughput, irregular data alignment, and program consistency. Ideally the match unit would be able to process the data as quickly as it arrives. However, there is a trade-off between circuit complexity and throughput. This trade-off led to an architecture with a small number of computational units operating in parallel. The need for high throughput also influenced the choice of a three stage pipeline depth of the unit. In general, the first cycle reads operands from the appropriate register file, the second cycle does the operation, and the third writes the result to the register file.

Inherent in header and list item processing is the necessity to process data of different bit widths packed into native size words, in this case 64-bits. This led to the inclusion of specialized functions to combine and reorder data, as well as the inclusion of SIMD-like functionality in the ALU and ternary unit. Finally, given the streaming nature of the data to the match unit, it is necessary to enforce strict ordering semantics for the program: all operations in the same instruction word are independent and their results are available for use by the next instruction. This is complicated by the fact that the input FIFO may become empty at any time. If the FIFO is empty and an element of the wide instruction requires input data, it is necessary to delay issuing the entire wide instruction until the FIFO has data. These issues make it necessary to include result forwarding (forwarding paths shown as dashed lines in Figure 3) and to require some of the register file write ports to be write before read. It is also necessary to modify the pipelining of the predicate unit (see Section 4.2.4).

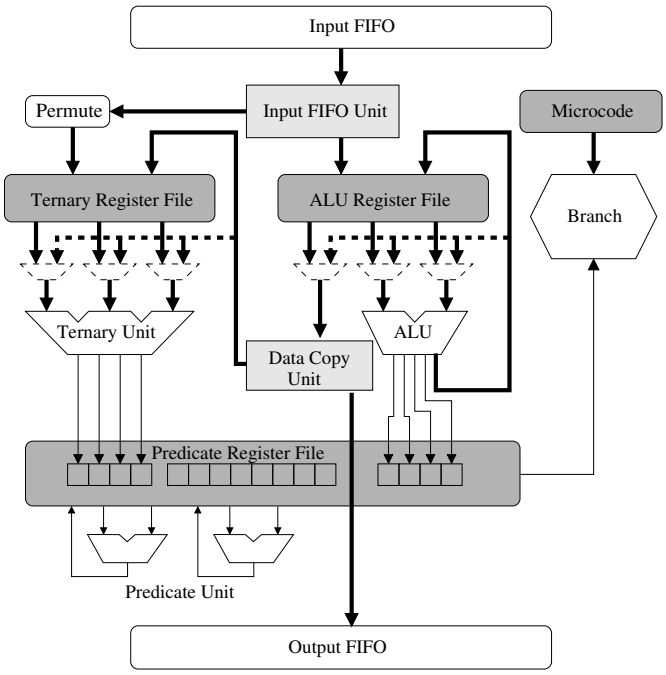

Figure 3. Block diagram of the microcoded match unit

4.2 Match Unit Details

The match engine consists of 4 computational/control units, 4 memories and 2 data transfer units, as seen in Figure 3. The computational units include the arithmetic logic unit (ALU), the ternary unit, the predicate unit and the branch unit. The four memories (shaded dark gray in Figure 3) consist of the microcode memory and three register files: the ALU registers, the ternary registers and the predicate registers. The data transfer units (shaded light gray in Figure 3) control data copies: 1) from the input FIFO to the ALU and ternary register files (Input FIFO Unit) and 2) from the ALU register file to the output FIFO or the ternary register file (Data Copy Unit). Each of the units has dedicated ports into the necessary memory elements allowing them to be controlled independently. This is done by giving each of the 6 units an instruction slot in the wide instruction word format of the microcode. The general format of the instruction word is shown in Figure 4. The bit widths of the major unit instruction fields are shown below the label for each field (the overall instruction word is 164 bits). The minor fields for each instruction are also shown. The following sections provide more detail about each of the major functional components.

4.2.1 Register Files

The ALU register file and the ternary register file are 64 bits wide and have 16 entries each. Each register file includes 2 write ports and 3 read ports. In both cases, register 0 is set to a constant, and thus, cannot be written. This constant is zero for the

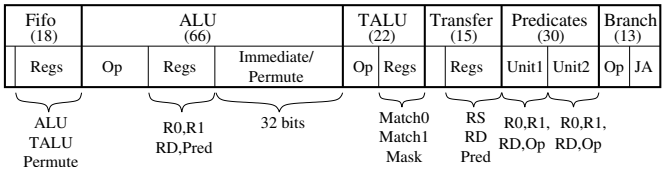

Figure 4. Format of the match unit's wide instruction word.

ALU and all 1's for the ternary unit. The connections to the read and write ports for these files can be seen in Figure 3. To comply with the program consistency semantics, the write ports connected to the input FIFOs are read before write and the other write ports are write before read.

The predicate register file contains sixteen 1-bit entries which can be accessed through 7 read ports. The write port structure is more complex: Eight of the registers are directly connected to the ALU and ternary unit to receive results of comparison operations. Register 0 is set to a constant 1, and the remaining 7 registers are writable from the predicate unit.

4.2.2 Arithmetic Logic Unit (ALU)

The ALU can perform most common binary operations, including addition, subtraction, logical operations and comparisons. Notably absent are the multiply and shift operations. Multiply is simply not needed in any known matching operations, but the removal of the shift operation may seem unusual. This decision was made as part of a trade-off between complexity and flexibility. Shifts are commonly used with logical operations to manipulate subsets of the ALU word size. Equivalent functionality is available from other components in the processor, so explicit shifts were removed. Bit level operations are supported by the ternary unit, as described in Section 4.2.3. Byte level (and above) operations are handled with the SIMD capability combined with an arbitrary permutation operation.

The permute operator can arbitrarily combine two registers on byte (8-bit) boundaries. This means that each byte of output can be chosen from any of the 8-bytes from either of the two inputs. In addition, each byte can also be set to all zeros. This function is needed to align the data fields in the header to the list item, and can replace a shifter for arbitrary byte level operations.

Because headers typically have a number of fields ranging from 16 to 64 bits, the ALU includes a feature to improve the efficiency of processing data smaller than 64-bits which are arbitrarily packed together into the native 64-bit words: the arithmetic functions are divided into 4 16-bit sections which can act in a SIMD fashion. Each segment executes the same instruction, but the segments can be aggregated together to perform larger functions. This is controlled by 3 SIMD bits, which tell the unit which internal 16-bit boundaries the operation will cross. Each 16-bit section also has its own comparison result output, and another set of 4-bits controls which of these results will be written to the predicate register file. This optimization allows the unit to work on the data when it is oddly aligned, and even allows it to operate on multiple fields simultaneously.

Because constants are periodically needed for various purposes in the match code, the instruction word for the arithmetic unit includes a 32-bit immediate field. This field doubles as part of the control field for the permute instruction. For all other instructions, however, this immediate can be injected as either the upper or lower 32-bits of the second operand. The remainder of the bits are passed through unchanged from the operand read from the register file (a traditional immediate instruction can be created by using register zero as the second operand). Since the immediate is the only mechanism for loading constants, creating a 64 bit constant involves replacing the lower 32 bits of a register and then replacing the upper 32 bits in a second instruction. For this reason, 64 bit constants that are needed frequently should be placed in a register at initialization (using two instructions) to save time and infrequently used constants should be built at execution time to save register file space.

To minimize the number of branch stalls needed, predication is used to control whether or not ALU operations write a result back to the ALU register file — a result is only written when its associated predicate is asserted. Operations that always write can use predicate register 0 (always true) as their predicate, and "nop" operations use ALU register 0 as their destination register. Unlike the other instructions, comparison operations do not write results to the ALU register file (and, therefore, are not affected by the predicate), but instead write their results directly to dedicated registers in the predicate register file. In addition to writing directly to the predicate register file, the results of comparison instructions can be combined with the existing predicate value as it is written (i.e. the new result can either overwrite the old result, or can be *and*ed or *or*ed with the old result). This is useful for quickly computing compound expressions. For example, checking to see if a value falls within a specified range (a $>$ 5 && a $<$10), or for checking to see if a field matches a particular value, or is set to accept any value (a == b || a == ANY). Although the compound functions could be computed in the predicate unit, the results are available a cycle faster when done during the register write. This is important given the relatively small number of cycles available for each match.

4.2.3 Ternary Unit

Unlike the arithmetic unit which has a variety of functions, the ternary unit performs one basic operation (although it can be used for multiple functions). As the name implies, there are three inputs to the unit: match0, match1 and mask. The ternary unit does an equal comparison under mask — only the bits which are specified in the mask are used in the comparison, all other bits are ignored[1]. Like the ALU, the ternary unit has SIMD functionality.

The ternary unit serves two primary purposes for the matching functionality. The first is to quickly see if the match bits in the header match the mask and match bits in the posted receive list (see line 11 in Figure 1. In addition, it can do equals comparisons on subsets of a word (i.e. smaller granularity than 16 bits). This second feature replaces the shifting and masking that would normally be used to pull out single bit flags from packed control words. This feature also allows for

[1]The unit can accept the mask as "ignore bits" or "don't ignore bits" depending on the mode

limited boolean functions (essentially wide *and* functions with optional negation of inputs) to be performed on flags which are found in the same word. For example, if a, b and c are one bit values packed into a single control word, the ternary unit could perform an operation such as `!a && b && !c`. As an additional benefit, the ternary unit can also be used as an equals comparison on any size field (for example, checking the expected NID with the NID in the header can happen in either the ALU or the ternary unit). The results are fed directly into the predicate register file, with the same facility for combining results described in Section 4.2.2.

The ternary unit also includes a simple permute unit on the input to the register file from the input FIFO. This permute divides the input into four 16-bit fields (corresponding to the four fields in a SIMD operation) and allows each of the 4 output fields to arbitrarily select any of the 4 input fields. This is useful for allowing the ternary unit to pull multiple flags out of a single 16-bit word. If this simplified permute is not sufficient, the more complex permute of the ALU can be used to properly stage the data.

4.2.4 Predicate Unit

The predicate unit is used to combine predicates generated by the arithmetic and ternary units. The unit consists of the predicate register file and two logic units which can each perform arbitrary boolean functions on two predicates. The two logic units can read from any register, but can only write to registers 1 through 7. There are also three other read ports: one each for the branch unit, the ALU, and the data copy unit. All of the read ports are write before read, allowing predicates to be read and used more quickly.

The predicate unit uses slightly different timing so that all units have consistent timing. The first cycle does nothing, the second cycle reads from the register file and the third computes and writes the result. This is possible because the predicates are only single bit values.

4.2.5 Branch Unit

The branch unit simply controls the flow of the microcode program. A branch instruction specifies a predicate register and an absolute target address. The branch instruction can then be either branch on one or branch on zero. Thus, a branch depends on the results of some number of previous ALU comparisons that have been accumulated in predicate registers (either by the ALU write accumulation capability or the predicate unit accumulations).

In the absence of a branch command, the unit simply retrieves the next instruction word. A branch requires two cycles to resolve, therefore, there are two branch delay slots. All instructions in these two slots will be executed on a taken branch, with the exception of other branch commands; a taken branch will invalidate branches in the next two instructions. These branch delay slots are generally easy to fill as most branches are “early out” cases where the header does not match that list item; thus, further list item comparisons can go in these slots.

Table 1. Breakdown of the assembly code

Code segment	Instructions
Initialization	4
Header	9
List Item (Shared)	6
List Item (Fast Path)	7
List Item (Slow Path)	10
Flush	8
Total	44

The branch unit also controls program flow when the input FIFO is empty. Each instruction can be tagged as requiring FIFO input or not. If the input FIFO is empty and the instruction requires input, then the branch unit will invalidate the current instruction and hold the program counter at its current value. Instructions which have already issued will continue to progress through the pipeline, with the exception of branch instructions, which will wait for input to proceed. This is required to ensure that both branch delay instructions execute.

4.2.6 Data Transfer Units

The match unit has two data transfer units to move data around the processor. The input FIFO unit simply takes data from the input FIFO and moves it to either the ALU register file, the ternary register file, or both simultaneously. Instruction words that contain a command for the input FIFO unit stall if the FIFO is empty (does not contain data).

The second data transfer unit copies from the ALU register file into the ternary register file or into the output FIFO. As with all instructions that write to register files, these instructions can be controlled by a predicate. Unlike the input FIFO unit, this data transfer unit cannot cause the overall instruction to stall, because the output FIFO can always accept data (never becomes full).

4.3 Matching Code Characteristics

The match unit is programmed entirely in an assembly language that translates one-to-one into microcode instructions. Part of the design target of the architecture was to minimize the number of instructions (and, therefore, cycles) required to implement the matching code. Table 1 gives a breakdown of the 44 instructions required to implement the matching operation. Initialization is needed to establish constants that will be used in the primary loop and is only executed at boot time. The primary loop for matching includes one execution of the header code, one or more executions of the list item code (typically shared code plus the fast path), and one execution of the flush code for a total of at least 31 cycles (a pipelining impact). Each additional list item traversed adds at least 8 cycles (assuming the common case). More detail on the individual code segments follows.

The header code must read an 8 item header (one per cycle) from the input FIFO. Because header fields and list item fields are not typically identical, this code must reformat the data to match the list items to improve matching speed. The list item code then reads list items and compares them to the header. This code is split into "fast" and "slow" paths that share a common preamble, where the "slow" path supports a less common Portals semantic on a per list item basis. The "fast" path is optimized to complete a list item as soon as a match fails, but this cannot be less than 8 total cycles, as it takes 8 cycles to read the list item from the FIFO. The most common match failure is the `match_bits` test that occurs first in Figure 1. This failure require 8 cycles per list item, where a full match will execute 12 instructions in 14 cycles (on a match, only 5 of the 6 "shared" instructions are executed). Finally, in most cases, the list manager will have sent an extra list item to be matched — not knowing that a match will be found. This requires that the flush code execute (8 cycles) to drain the extra list item from the input.

5 Methodology

We compare the matching performance of three basic architectures: the customized architecture described in this paper, a typical embedded CPU and a multithreaded CPU. The more traditional processors were simulated using the Structural Simulation Toolkit (SST)[25], and are described further below; however, the microcoded match unit was done in a cycle accurate hardware simulator.

5.1 Benchmark

The performance of the three architectures is compared using a benchmark that times the match under different conditions. The benchmark measures the total match time for 64 matches based on the total length of the posted receives queue, as well as the number of list items actually traversed. The core operation of the benchmark is shown in Figure 5. The `match()` function is shown in Figure 1. Since the benchmark is designed to only measure match time, the code does not delete anything from the list, it just performs the match.

In the benchmark code, the outer loop goes through a set of headers which are designed to match 0% (meaning the first entry) through 100% of the way through the list in increments of 10%. In addition, the first iteration is used to prime the instruction cache so that there are no misses during execution. The inner loop reads 64 identical headers and is the only part included in the measured time. A large number of headers was chosen to allow for a comparison with the multithreaded unit, which provides no advantage for a single header, but which can greatly improve throughput for a large number of rapidly received headers. To support the multithreaded processor, two specific changes to the code were made. First, each call to `match()` was invoked in a new thread. Second, locking was added to the `match()` function to insure that two different threads did not simultaneously access one list item and to insure that one thread did not "pass" another thread and cause out

```
for ( i = 0; i < 12; i++ ) {
  struct memd *match_memd;
  struct ptlhdr *start = (ptlhdr *)(offset);

  int before = readSimCycle();
  for ( j = 0; j < 64; j++ ) {
    start = getNextHeader();
    match_memd = match(list,start,0,&offset,&length);
  }
  int after = readSimCycle();
  times[i] = after-before;
}
```

Figure 5. Core operation of match performance benchmark.

Table 2. Embedded Processor Configuration

Fetch/Decode Width	2
Issue Width	2
Commit Width	3 inst.
RUU Size	16
LSQ Size	4
dL1 Cache	32KB 64-Way
iL1 Cache	32KB 64-Way
dL2 Cache	None
Latency L1/memory	1/10 cycles
Int. ALU	2
Int Mult	1
Memory BW (bits/cycle)	128
Memory Ports	1

of order matching to occur. Both thread startup and locking are simulated as exceptionally fast to insure that the comparison to multithreaded processor is not unfair. Details about each configuration are given below.

5.2 Embedded CPU

The simulation model for the embedded CPU is configured to be similar to a PowerPC 440 processor, as shown in Table 2. The SimpleScalar[7] processor model was used to simulate the PowerPC using a PowerPC instruction set. The code was compiled with gcc 3.3.3 and targeted Mac-OSX (the loader supported by the simulator — no OS was used).

5.3 Multithreaded CPU

Table 3 highlights many of the properties of the multithreaded processor configurations. The number of execution cores was varied for typical (1 and 2 core) configurations as well as an aggressive configuration reflective of a state of the art

Table 3. Multithreaded Processor Configuration

cores	1	2	16
Fetch/Decode Width	1	1	1
Issue Width (in-order)	1	1	1
Commit Width	1	1	1
dL1 Cache (64-Way)	32KB	16KB	16KB
iL1 Cache (64-Way)	32KB	16KB	16KB
dL2 Cache	None	None	None
Latency L1/mem. (cycles)	1/10	1/10	1/10
Int. ALU (inc. Mult)	1	1	1
Threads/core	8	8	4
Memory BW (bits/cycle)	128	128	256

multithreaded network processor like the Intel IXP2800[12]. Each execution core had a fixed number of hardware thread contexts, of which up to 64 total contexts could be used by the benchmark[2].

The thread creation time was only two cycles. While this is aggressive, it is also conceivably possible in this type of application; thus, we chose this design point to make the multithreaded core as competitive as possible. Each core switches between active contexts each cycle. Every 1024 cycles (or, when there are no active contexts), inactive contexts are swapped into the core. Inactive contexts are created whenever a new thread is spawned and there is not a free hardware context to hold it. Swapping in an inactive context takes 10 cycles. Again, this is an aggressive design point, but it is chosen to maximize the competitiveness of the multithreaded architecture.

5.4 Microcoded Match Unit

The microcoded unit is simulated using cycle accurate hardware simulation in JHDL[10]. The match code was hand coded in assembly code and translated to microcode to be run on the match unit. The simulation assumes that the list items are read by the list manager as discussed in Section 3. Since the list manager is separate from the match code, it is likely that the match unit will receive an extra item after a match is found. The match time includes the time required to flush this extra item and notify the list manager.

6 Results

We selected four technology points for comparison: a conventional embedded processor, a multi-core, multi-threaded processor, a pure hardware unit with memory bandwidth to match the microcoded match unit, and our proposed match unit architecture. These were selected to represent current practice in NICs supporting MPI, current state of the art NPUs, the "best case"[3], and our proposed design. We present data assuming that each architecture is running at 500 MHz under the

[2]This also happens to be the maximum number of threads in any context.

[3]This is only a "best case" for the bandwidth available to our microcoded match engine and is selected because a wider (higher bandwidth) hardware unit would actually pose significant implementation challenges.

assumption that each design point could approach approximately the same clock rate with sufficient effort.

6.1 Performance

Figures 6 and 7 present comparative data for the various processor options at four different list lengths: 10, 30, 100, and 300 items. The data is presented in three different ways: as an absolute time (a, b) , as a time per list item traversed (c, d), and as the time relative to a best case hardware unit (e, f). The best case hardware time assumes that list items can be processesed as quickly as they can be fed to the matching unit; thus, is assumes a flat overhead of 8 cycles to load the header and a flat delay of 8 cycles per item traversed. In all cases, the X-axis is the percentage of the list that must be traversed to find a match and the Y-axis is a metric of time. In general, the configurations considered here do not encounter the caching effects seen in [23].

The most notable property of all of these graphs is that all of the programmable configurations pay a larger fixed overhead than a pure hardware implementation. This is particularly noticeable for the embedded processor and threaded processors, where the memory latency for loading the header imposes a significant overall penalty when a single list item is traversed. As more list items are traversed, this overhead is amortized away. The overhead for the microcoded match unit is only 6 cycles per *item matched* with no penalty per *item traversed*. The 6 cycle penalty is the difference between the time to match an item using the microcoded match unit (14 cycles) and the time to feed an item to be matched into the match unit (8 cycles). For our streaming test, this results in a constant 384 cycle penalty (64 incoming items that match in the list, 6 cycles per item that matches). The embedded processor clearly pays a penalty (relative to the hardware approach) for both each *item traversed* as well as each *item matched*, because the "zero length" time is larger than the asymptotic time per item and the asymptotic time per item does not approach the hardware limit. The multithreaded units, however, exploit much more concurrency with the multicore cases so that the asymptotic time per item approaches the lower bound of the hardware time as the list grows long. They do, however, pay a higher *item matched* penalty.

At short list lengths (10 items), we see nearly a $2\times$ advantage for the microcoded match unit over any of the other configurations — an advantage that grows to $3\times$ if only a portion of the list is traversed. In general, the embedded processor, with its more robust pipeline and out-of-order execution capabilities, has a significant win (14%) over the the largest multithreaded configuration when traversing only a few items. However, even when traversing only 10 items, the concurrency that the multithreaded unit is able to exploit yields a slight advantage for the single multithreaded core and a 34% advantage when 16 multithreaded cores are used.

As the list length grows (30 items), the multithreaded scenarios begin to distinguish themselves from the embedded processor. Although there are significant impacts from computation time, memory latency impacts are sufficient to make the latency tolerating strengths of the multithreaded cores evident. With a list of only 30 items, however, there is still not sufficient concurrency to dramatically differentiate 16 multithreaded cores from 2. In all of these cases, the microcoded

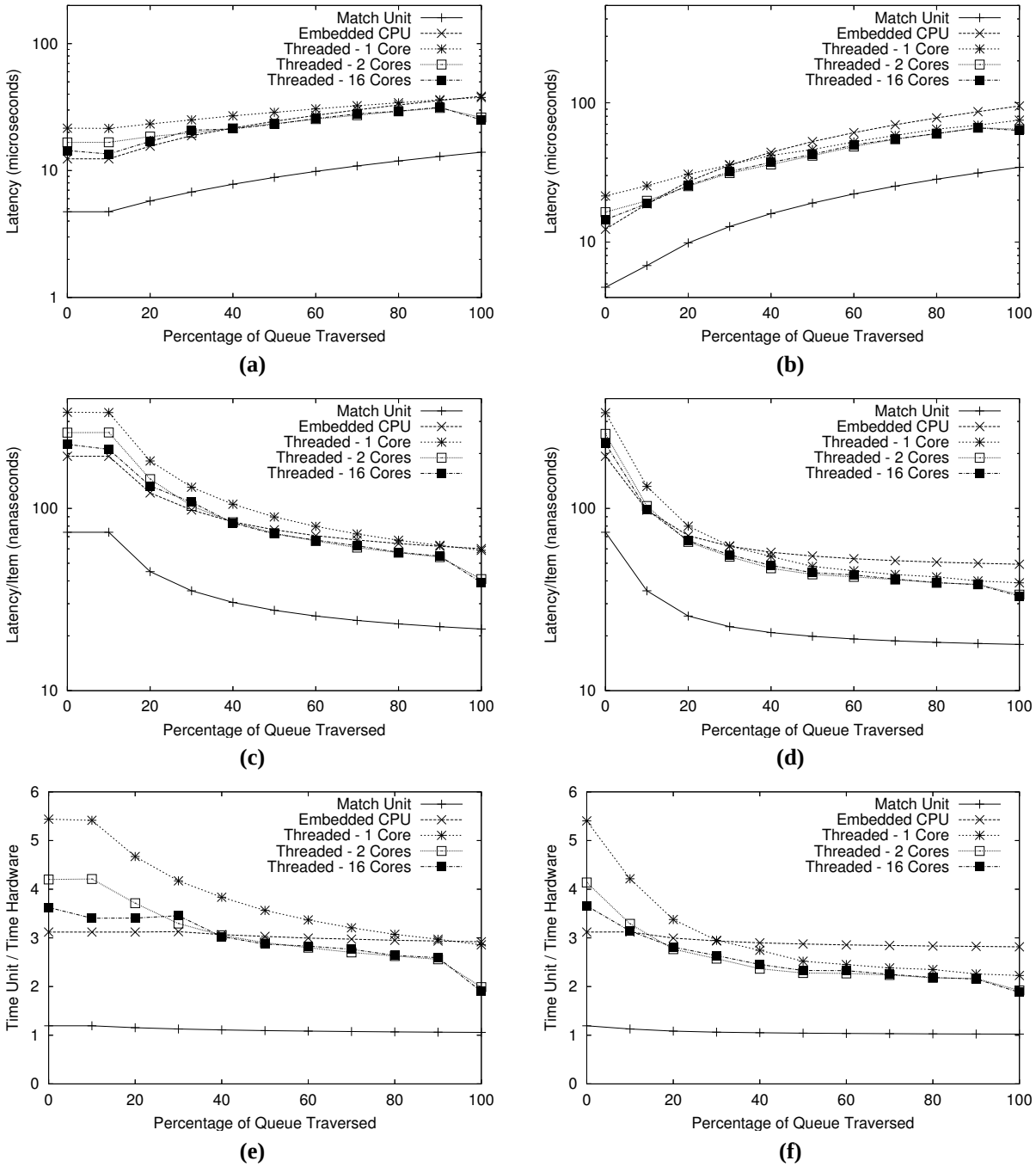

Figure 6. Total time for (a) 10 item list and (b) 30 item list; time per item for (c) 10 item list and (d) 30 item list; time relative to hardware for (e) 10 item list and (f) 30 item list

Table 4. Estimated Sizes of Studied Architectures

	Match Unit	PPC 440	Multithreaded		
Cores	1	1	1	2	16
Area (mm^2)	1.3	6	5	10	80
Relative Area	$1\times$	$4.6\times$	$3.8\times$	$7.7\times$	$61.5\times$

match unit maintains a dramatic advantage over the most aggressive of the multithreaded configurations — an advantage of almost $2\times$! In the case of the microcoded match unit, the overhead over a pure hardware solution (which begins at 19%) has dropped to only 2.3% when 30 list items are traversed.

As the list grows long, sufficient concurrency becomes available to clearly highlight the hardware advantages of adding more multithreaded cores. Where the absolute gap between the embedded microprocessor and the microcoded match unit is a virtually constant value across a range of list traversal lengths, adding more items to the list increases the available concurrency and offers the multithreaded cores an opportunity to shine.

With 100 list items traversed, the 16 multithreaded core case approaches the performance of the microcoded engine (although using drastically more hardware). By the time the list reaches 300 items, 2 multithreaded cores approach the performance of the microcoded match unit and 16 multithreaded cores exceed it. Indeed, leveraging the fact that it is configured with $4\times$ more memory bandwidth than the microcoded match engine, the 16 multithreaded core case exceeds the "best case" scenario posed by the hardware at lower bandwidth. This is because 16 functional units are employed to work on 64 different incoming messages traversing a single list.

6.2 Area

A significant advantage of the proposed microcoded match unit is its savings in chip area compared to other potential approaches. Using the CACTI tool[20], we estimated that the memories in the match unit take $0.326mm^2$ in 90 nm technology. To make a very conservative estimate of the area required, we double this estimate ($0.652mm^2$) to account for the size of the functional units. In addition, although it should be much smaller, we assume that the list management unit needed to support the match unit is as large as the match unit itself for a total of $1.3mm^2$. In Table 4, we compare this area to that of an embedded processor (the PowerPC 440 approximated by our simulations[11]) and a multi-core, multithreaded approach. For the multithreaded approach, we leveraged a die photo and area information about the Sun Niagara multithreaded processor found in [13] and then used information available about the IXP2800[14] as a sanity check[4].

The microcoded match unit has significant area advantages over the single core approaches and dramatic advantages over the multicore approach (16 multithreaded cores) that is actually competitive in performance. The primary area advantage comes from the fact that neither the match unit or the list manager has a large cache integrated with it. While it could be

[4]The Niagara information was much more complete than the IXP2800 information.

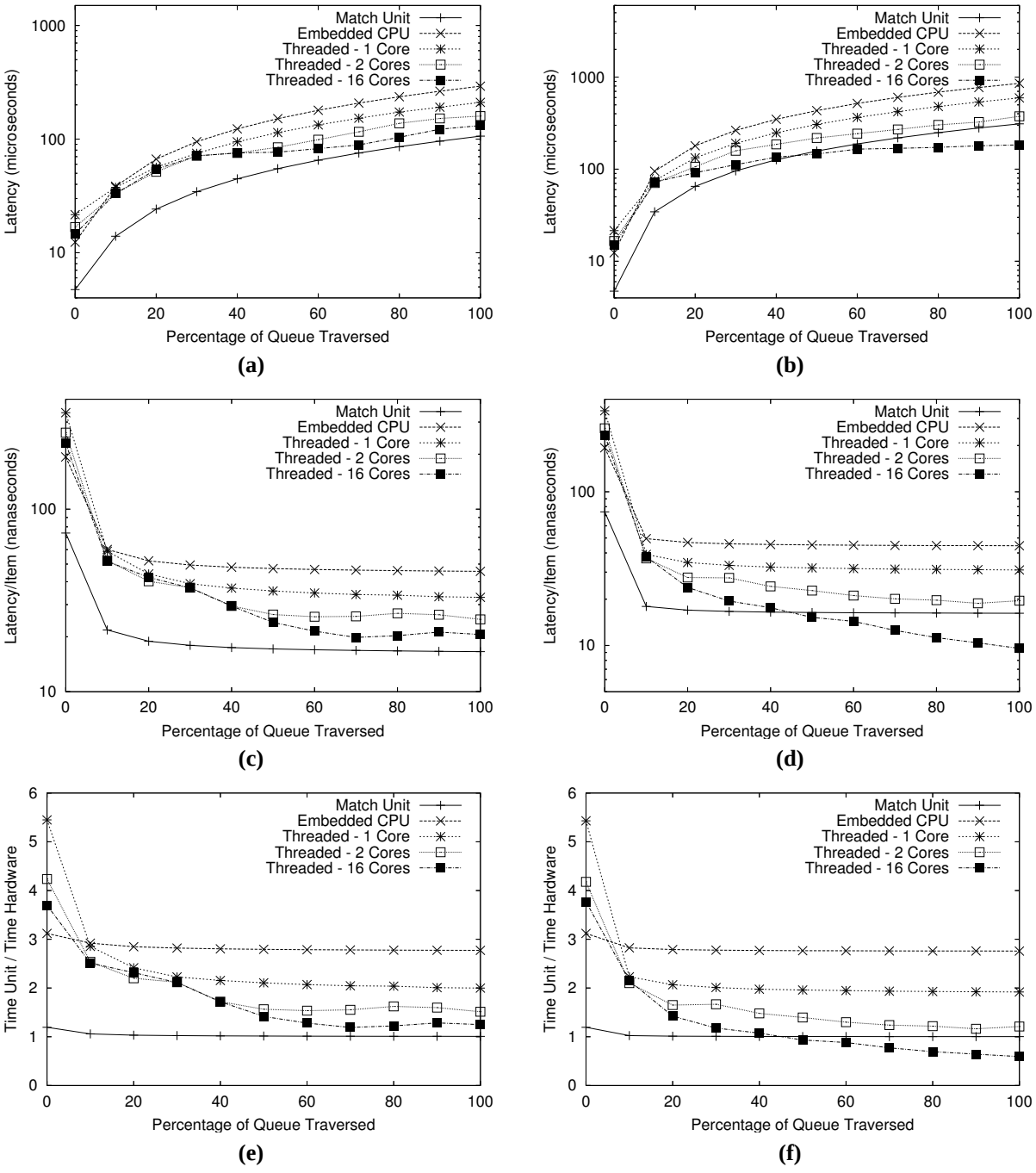

Figure 7. Total time for (a) 100 item list and (b) 300 item list; time per item for (c) 100 item list and (d) 300 item list; time relative to hardware for (e) 100 item list and (f) 300 item list

argued that the cache could be eliminated from the more conventional processor designs, these designs have these caches by default and if a new processor design is going to be created, it is better to optimize the architecture to the domain. Furthermore, the performance of both the embedded microprocessor and the multithreaded cores *depend* on that cache. In the case of the embedded microprocessor, the cache is needed to hide the memory latency. In the multicore multithreaded approach, the cache acts as an effective memory bandwidth multiplier to make it feasible to design the system.

6.3 FPGA Prototype

The architecture was mapped onto a Virtex4 (-11 speed grade) FPGA to get approximate operating frequency. The prototype was able to operate at 150MHz on the FPGA, which is produced using a 90nm CMOS process. Conservative estimates place standard cell ASICs at $5\times$ the clock rate of FPGAs; thus, we would expect the design to easily achieve a 750MHz operating frequency in a 90nm standard cell ASIC.

7 Conclusions

As supercomputer networks are pushed to ever lower latencies and ever higher message rates, it will become necessary to perform MPI matching at increasingly high rates. Rather than rely on a pure hardware implementation, we present a customized architecture to perform the MPI matching operation: the microcoded match unit. The customizations include the elimination of the traditional memory interface in favor of streaming data to be matched in through FIFO based constructs. In addition, the architecture includes two ALUs — one of which can only perform ternary operations — that are both capable of multiple simultaneous sub-word operations that match the irregular data structures typically found in network headers and linked list elements. Finally, the architecture includes a high degree of concurrency that enables six types of operations in each cycle.

We compare the proposed architecture to an upper bound on performance formed by a dedicated hardware implementation and find that the microcoded match unit is within 20% of this upper bound when only a single item is traversed and within 6% of this bound when 10 list items are traversed. In contrast, we also compare the microcoded match unit to a conventional embedded processor and a multi-core multithreaded approach. We find that the microcoded match unit is almost $3\times$ faster than either when the list is only a single element long and over $2\times$ as fast when the list is 10 items. In fact, the multithreaded approach only approaches comparable performance when the list is 100 items long and only exceeds the performance of the simple microcoded match unit when there are 16 multithreaded cores and the list is hundreds of items long. These results were achieved while using an area that is $4.6\times$ smaller than an embedded microprocessor and $3.8\times$ smaller than a *single* multithreaded core.

8 Future Work

The core of the matching algorithm is the key place where flexibility is required in the overall MPI processing pipeline; however, we plan to continue to explore microarchitectural requirements for other portions of the pipeline as a mechanism to provide flexibility without sacrificing performance. A key example is the management of the posted receive queue. While a parameterizable hardware unit can be built to manage a simple list, using a programmable unit makes it easier to change the list format, implement improved free space management algorithms, or cope with hardware bugs elsewhere in the system.

References

[1] M. J. Akhbarizadeh and M. Nourani. Efficient prefix cache for network processors. In *Proceedings of the 12th IEEE Symposium on High-Performance Interconnects*, August 2004.

[2] R. Brightwell, T. Hudson, K. T. Pedretti, and K. D. Underwood. SeaStar interconnect: Balanced bandwidth for scalable performance. *IEEE Micro*, 26(3), May/June 2006.

[3] R. Brightwell, T. B. Hudson, A. B. Maccabe, and R. E. Riesen. The Portals 3.0 message passing interface revision 2.0. Technical Report SAND2006-0420, Sandia National Laboratories, January 2006.

[4] R. Brightwell, W. Lawry, A. B. Maccabe, and R. Riesen. Portals 3.0: Protocol building blocks for low overhead communication. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters*, April 2002.

[5] D. Buntinas and D. K. Panda. NIC-based reduction in Myrinet clusters: Is it beneficial? In *Proceedings of the SAN-02 Workshop (in conjunction with HPCA)*, February 2002.

[6] D. Buntinas, D. K. Panda, and P. Sadayappan. Fast NIC-based barrier over Myrinet/GM. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2001.

[7] D. Burger and T. Austin. *The SimpleScalar Tool Set, Version 2.0*. SimpleScalar LLC.

[8] C. Clark, W. Lee, D. Schimmel, D. Contis, M. Kone, and A. Thomas. A hardware platform for network intrusion detection and prevention. In *Proceedings of the Third Workshop on Network Processors and Applications*, Madrid, Spain, February 2004.

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[10] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting. A CAD suite for high-performance FPGA design. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 12–24, Napa, CA, April 1999. IEEE Computer Society, IEEE.

[11] IBM. Ibm powerpc 440 embedded core product brief. http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/F72367F770327F8A87256E63006CE Nov. 2006.

[12] Intel Coporation. *Intel IXP2805 Network Processor*, 2005.

[13] P. Kongetira. A 32-way multithreaded sparc processor. In *Hot Chips 16*, August 2004.

[14] Y.-K. Lai and G. T. Byrd. High-throughput sketch update on a low-power stream processor. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 123–132, New York, NY, USA, 2006. ACM Press.

[15] R.-T. Liu, N.-F. Huang, C.-H. Chen, and C.-N. Kao. A fast string-matching algorithm for network processor-based intrusion detection system. *Trans. on Embedded Computing Sys.*, 3(3):614–633, 2004.

[16] Message Passing Interface Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.

[17] A. Moody, J. Fernandez, F. Petrini, and D. K. Panda. Scalable NIC-based reduction on large-scale clusters. In *Proceedings of the ACM/IEEE SC2003 Conference*, November 2003.

[18] Myricom, Inc. Myrinet Express (MX): A high performance, low-level, message-passing interface for Myrinet, July 2003.

[19] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, January/February 2002.

[20] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. WRL Research Report 2, Western Research Laboratory, 250 University Lane, Palo Alto CA, 2001.

[21] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven gigabit ethernet message passing. In *Proceedings of the 2001 Conference on Supercomputing*, Nov. 2001.

[22] J. Singaraju, L. Bu, and J. A. Chandy. A signature match processor architecture for network intrusion detection. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, California, April 2005.

[23] K. D. Underwood and R. Brightwell. The impact of MPI queue usage on message latency. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, Montreal, Canada, August 2004.

[24] K. D. Underwood, K. S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell. A hardware acceleration unit for MPI queue processing. In *19th International Parallel and Distributed Processing Symposium (IPDPS'05)*, April 2005.

[25] K. D. Underwood, M. Levenhagen, and A. Rodrigues. Simulating Red Storm: Challenges and successes in building a system simulation. In *21st International Parallel and Distributed Processing Symposium (IPDPS'07)*, March 2007.

[26] K. D. Underwood, A. Rodrigues, and K. S. Hemmert. Accelerating list management for MPI. In *Proceedings of the 2005 IEEE International Conference on Cluster Computing*, September 2005.