Prefetching Across a Shared Memory Tree within a Network-on-Chip Architecture

Jamie Garside
Department of Computer Science
University of York, UK
jg@cs.york.ac.uk

Neil C. Audsley
Department of Computer Science
University of York, UK
neil.audsley@york.ac.uk

***Abstract*— Within Network-on-Chip architectures the sharing of external memory by many CPUs provides a key challenge within the design in order that memory latencies do not dominate overall performance. Within this paper, we propose and evaluate a stream based prefetch unit within a NoC architecture that utilises a separate shared memory tree to provide access to external memory from each CPU tile. The paper shows that prefetching is an appropriate architectural technique within NoCs, enabling better system performance.**

I. Introduction

The gap between the relative performance of a CPU and its attached memory is continually widening [1]. In the context of Network-on-Chip (NoC) architectures with multiple CPUs and shared external memory, these effects are magnified as external memory accesses also involve routing latencies across a grid of routers. In single CPU systems, caches and prefetch are typical techniques to hide memory latencies. This paper proposes and evaluates some prefetch techniques within a NoC architecture using CPUs with caches enabled, accessing external memory across a dedicated shared memory tree.

The remainder of this paper is structured as follows. Section II will provide an overview of previous work within the field of prefetching. Section III will describe the network-on-chip we are using for our experiments, while V will detail the methodology of evaluating a prefetcher within the context of the network-on-chip. Finally, evaluations of these experiments will be given in section V and conclusions drawn in section VI.

II. Background and Related Work

Stream prefetching [2] makes the simple assumption that if memory blocks at addresses $n-2$, $n-1$ and n have been accessed in sequence, it is likely that the processor will soon require block $n+1$. This can be implemented using a lookup table indexed on the last miss address for the stream corresponding to a table row. On a cache miss, the miss address is looked up in the table, the table updated, and possibly the next line fetched from memory. On a cache hit, if the cache line is tagged as prefetched, the prefetcher is notified, which will again look up the address in the table, update it, and issue a request for the next line. If there is no row corresponding to a cache miss address, a different row is picked as a replacement candidate, typically using LRU or round-robin selection, and filled in with the information of the miss address.

Stream prefetching can only detect and prefetch a subset of all possible traffic patterns, but is ideal for code prefetching. To detect streaming data with an arbitary distance d between each block, [3] developed a prefetcher which if blocks $n-2d$ and $n-d$ have been requested, the processor will require blocks n and $n+d$ in the near future. This is implemented by a lookup table indexed by the program counter rather than the miss address. A stride can be detected without many complex lookups. However, for a large loop kernel with many loads, there must be a sufficiently large table in order to store all of this information. [4] shows that for program counter-based approaches such as this, a table of around 256 elements is required, any less starts to greatly harm performance.

For data accesses that are more unpredictable in terms of stride distance differences, two main approaches have been proposed. For every memory address causing a cache miss, Markov prefetching [5] stores which memory addresses came next in the reference stream and prefetches those addresses too. This requires a large amount of storage space to encode this table, so a generalisation of this is to instead store the differences between the memory addresses rather than the addresses themselves [6]. This then can provide similar performance to a Markov prefetcher, albeit with more potential pollution, at a fraction of the space required by the Markov model. Other approaches either examine data fetched from memory to find memory addresses (i.e. pointers to data in a linked data structure) [7], [8] or require the programmer to annotate their data structures with prefetch candidates [9].

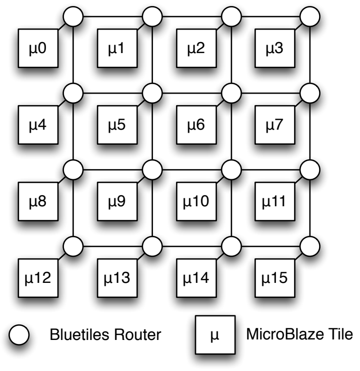
Prefetchers such as these can be tuned on a number of different parameters, typically on the amount of data fetched at once (the prefetch degree), and how far ahead of the current miss address data is fetched (the prefetch distance). Typically, these are set at design time based upon experiments on the access patterns the prefetcher is expecting to see.

III. System Architecture

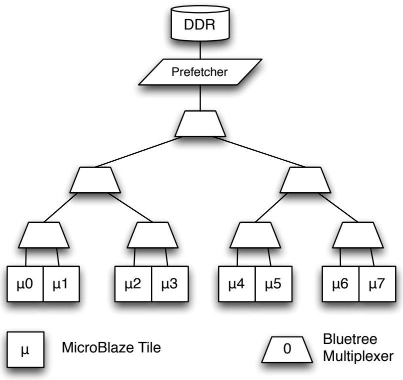
This paper uses the Blueshell NoC framework [10], whose basic architecture is shown in Figure 1(a) and 1(b). Specific configuration of the framework for this paper includes:

- *Routers*: are 32-bit bi-directional. Routing is performed by inspection of the first word for destination, with X-Y

This work is supported in part by EU FP7 project T-CREST (288008)



(a) Bluetiles NoC with 16 CPU Tiles



(b) Bluetree Shared Memory Tree for 8 CPU Tiles

Fig. 1. Blueshell Network-on-Chip and Shared Memory Tree
Bluetiles is a Manhattan grid NoC, with arbitrated routing between CPU tiles. Each CPU tile also connects directly (via cache) to the Bluetree shared memory (binary) tree to access external DDR. There is no interference between CPU to CPU messages across the NoC and CPU to memory transactions across the tree. CPU tiles can be replaced with other custom accelerators or I/O to peripherals.

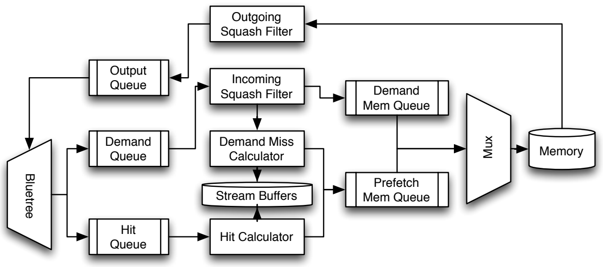


Fig. 2. Architecture of the Prefetch Unit

routing then performed. Within the Bluetiles framework there are a number of potential routing policies that can be chosen – we note that the exact choice does not impact upon the research presented in this paper.

- *Shared Memory Tree*: 2-to-1 multiplexors are used, connecting all CPU tiles at the leaves of the tree and the memory at the root. High-bandwidth memory requests do not impact the performance of other CPUs – there is no interference between CPU to CPU messages across the NoC and CPU to memory transactions across the tree.
- *CPU Tiles*: Microblaze CPU tiles are used, with 8kB of local storage, 8kB split data and instruction caches, and a 8kB shared scratchpad used for fast local storage. The CPU interfaces to the shared memory tree via Microblaze FSL links (single cycle FIFOs), allowing cache misses to be issued to external memory via the shared tree. Custom cache control is configured to allow selective invalidation of cache lines and to record prefetch related data on a per cache line basis.

A. Prefetch Unit (PU)

This paper proposes a streaming Prefetch Unit (PU) at the root of the shared memory tree (see Figure 1(b)) to snoop all memory transactions to ascertain trends in the pattern of memory fetches, and overall load of main memory. The PU is variable lookahead, in that it can be configured to distance 1, 2 or 4.

The architecture of the PU is illustrated in Figure 2:

- *Stream Buffer*: 8 per connected CPU. They store last address accessed and validity of the stream. Each stream buffer is circular.
- *Prefetch Buffer*: 32 entry circular buffer containing all prefetches currently issued to memory (ie. pending).
- *Squash Buffer*: 32 entry circular buffer containing all pending requests from CPUs that should be coalesced with an outstanding prefetch.

Both the Prefetch and Squash buffers are shared, but have single cycle access eliminating contention issues.

The operation of the PU is described in Listing 1. Additionally we note the following:

a) Cache Miss: if the address is not in the Prefetch Buffer, a memory request is issued and a prefetch of the next memory location (plus distance D) instigated. If the address is already being fetched (ie. in the Prefetch Buffer), an entry is made in the Squash Buffer and the request discarded. The cache marks the returned memory packet as *prefetched* by asserting a flag within the cache entry.

b) Cache Hit: on a cache hit of an address marked *prefetched* the PU is informed so that it knows the prefetch was useful using a special Bluetree packet sent automatically from the cache, so can instigate the next prefetch for this stream.

c) Potential Race Condition: a demand miss (i.e. a block required by the CPU) could arrive for a prefetched block which is currently being delivered to the CPU via the shared memory tree. Hence, on a completed prefetch, the corresponding line in the prefetch buffer is set as *recent* rather than *invalid*. If a line is found in the prefetch buffer which is marked as *recent*, the memory request can be discarded, since the recent prefetch will fulfill the demand miss. In order to avoid hazards, any line in the prefetch buffer which is marked *recent* can be overwritten by new prefetches, or overwritten if there is a

```
On incoming memory request:
  if(request_address in prefetch_buffer):
    add request_address to squash_buffer
    discard request
  else:
    issue request_address to memory
    if(request_address-D in stream_buffers):
      issue request_address+D to memory
      update stream_buffers
    else
      add request_address to stream_buffers

On incoming hit notification:
  if(hit_address in stream_buffers)
    issue hit_address+D to memory
    update stream_buffers

On returning memory request:
  if(return_address in squash_buffer)
    send return_packet to cpu as standard read
  else
    send return_packet to cpu as prefetch
```

Listing 1. Pseudocode describing the operation PU (distance D).

write to that memory line.

IV. Implementation

The streaming Prefetch Unit (PU) (see section III-A) was implemented within a 4x4 Bluetiles NoC (see section III), with each CPU having a connection to the shared memory tree. The PU is at the root (see 1(b)), hence a depth of 4 multiplexors between any CPU and the PU, and a further multiplexor between PU and DDR (not illustrated in 1(b)). The target for the implementation was a Xilinx Virtex-7 FPGA attached to DDR3-800 external memory, utilising the standard Xilinx DDR3 controller [11]. The CPUs are configured to run at 50MHz[1], and are synthesized onto the physical hardware.

In terms of timing and latencies, we note:

- *Shared Memory Tree Multiplexor*: Bluetree uses a buffer for both input and output, hence the cost of crossing a multiplexor is 2 cycles. Note that the multiplexor between PU and DDR crosses a clock domain, with total cost $\approx$15 clock cycles.
- *PU*: cost of crossing is 2 cycles.
- *DDR*: clocked at 100MHz, with a $\approx$ 10 cycle setup and transfer time for a memory transaction.

Therefore, the total latency for a memory transaction from CPU to PU and return is 20 cycles (including crossing the PU), between PU and DDR and return is $\approx$ 40 cycles. Hence total time is $\approx$ 60 cycles.

V. Evaluation

The Prefetch Unit (PU) was evaluated via CPUs executing a traffic generator fetching from addresses N, $N{+}1$, $N{+}2$ etc in sequence, waiting between accesses in order to simulate varying memory traffic load. Delays between successive memory requests from the CPU were set at 300, with memory speed and load measured at the PU. Then separate experiments were performed for each delay from 300 down to 0 cycles (step 3 cycles). Zero-delay stresses the memory controller as much as possible. With a delay of 300 the time spent in the delay loop will grow much larger than that for memory requests, and as such the speedup will tend to zero. The number of enabled CPUs were also varied (between 1 and 15); together with the prefetch distance.

For each experiment, two metrics are recorded:

- *Normalised Execution Time*: representing the potential speedup of prefetching. This is the ratio of the amount of execution time achieved by the CPU when the PU is active to that when the PU is bypassed (ie. a conventional system without a PU). A normalised execution time of 1 represents no speedup, 2 represents 100% speedup, and $<$1 represents a performance penalty.
- *Memory Load*: representing the proportion of time that there was an outstanding memory request (assuming the PU is bypassed, representing a conventional system without a PU). In general, as the delay is lowered, the memory load will increase.

A. 1-Lookahead PU with 16 CPUs

An upwards curve can be seen in figure 3(b). As delay between successive traffic requests increases, the memory load decreases due to the delays. As this delay increases further, the amount of time spent accessing memory eventually is dominated by the delay until the speedup tends to zero.

Also, PU speedup tends to zero as the memory becomes highly loaded, as prefetches have lower priority than memory requests demanded by the CPU (ie. cache misses). If the memory is highly loaded, the prefetches will not be dispatched in a timely manner and thus will coalesce in almost all cases. Finally, if the memory is so highly loaded that prefetches simply cannot be dispatched fast enough, stale prefetches will be served out of the prefetch queue.

B. 1-Lookahead PU with 8 CPUs

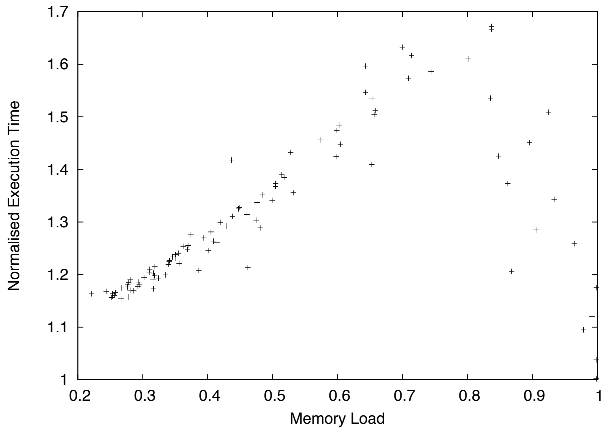
An upwards curve is seen in figure 3(a), similar to the results for 16 CPUs, although with a higher total speedup factor due to fewer CPUs sharing memory bandwidth. Comparing between 8 and 16 CPUs at 80% load point, the former has 20% of available bandwidth between 8 CPUs; the latter shares it between 16 CPUs.

C. Varying Lookahead PU with 4 CPUs

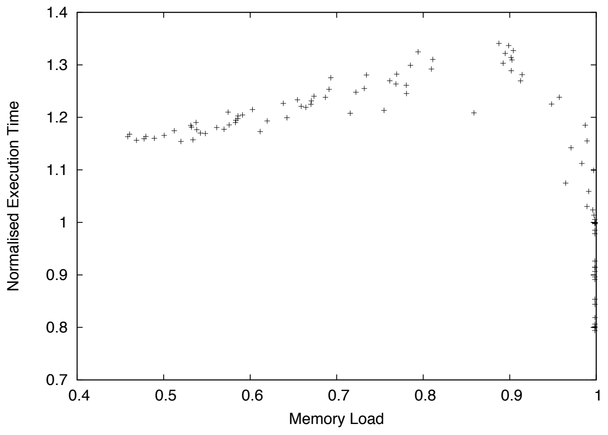
Figure 4 shows the best fit curves for prefetch distances 1,2 and 4-ahead. As the memory utilisation increases, the performance of a 1-lookahead PU falls quickly – this does not happen with 2-lookahead and 4-lookahead PUs since as the delay between memory accesses decreases, the next word will be required faster, leading to a coalesced prefetch for the next line. If instead a prefetch is issued for two words ahead, this coalesce does not occur, since the line now being prefetched is further ahead of the memory access stream.

We also note that as the delay decreases, the amount of free memory load for prefetches decreases due to overall memory

[1]We note that this is a tool-imposed restriction, and is being improved for future work. This low speed is not an issue for this work, since a set of CPUs can still saturate the available memory controller bandwidth.



(a) 8 processors



(b) 16 processors

Fig. 3. Plots of speedup against memory load for varying processor counts.

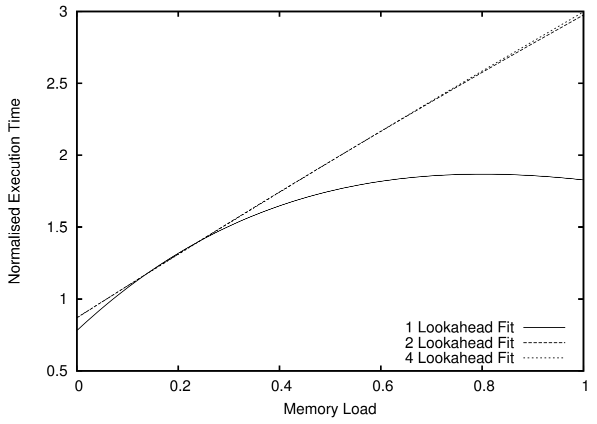


Fig. 4. Best fit curves for 4 Processors with multiple distances

load increasing. This leads to prefetches being dispatched later as standard memory packets take priority – hence they are more likely to coalesce with a demand miss.

VI. Conclusions and Future Work

Standard prefetch techniques, to an extent, can work on a multi-core system such as a NoC, masking a proportion of the increasing memory and bus latencies. This is particularly relevant as NoC sizes scale. We observed that larger prefetch distances can yield better results as the memory load increases. However, keeping the distance high does not always gain the best performance – eg. if the stream is short, or if unneeded prefetches are generated at the end of a stream (up to the prefetch distance).

Current work is extending the Prefetch Unit described with existing adaptive prefetch approaches. Srinath et al [12] propose a scheme which adapts the parameters of the prefetcher its current characteristics, i.e. the timeliness and accuracy of prefetches. Since the prefetcher in this system has global information about the memory load, it could adjust the prefetch distance based upon this load, since further prefetches are more likely to generate a hit, and shorter prefetches are likely to be shadowed due to the dispatch delay. Equally, at extremely high memory loads, the prefetcher should be disabled entirely, since it tends to only damage performance. Such global information will help develop a PU with increased effectiveness for NoC.

References

[1] J. Hennessy and D. Patterson, *Computer Architecture : A Quantitative Approach*, 4th ed. Morgan Kaufmann, 2006.

[2] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*. IEEE Comput. Soc. Press, 1990, pp. 364–373.

[3] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing - Supercomputing '91*. New York, New York, USA: ACM Press, 1991, pp. 176–186.

[4] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," *ACM SIGMICRO Newsletter*, vol. 23, no. 1-2, pp. 102–110, Dec. 1992.

[5] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in *Proceedings of the 24th annual international symposium on Computer architecture - ISCA '97*. New York, New York, USA: ACM Press, 1997, pp. 252–263.

[6] K. Nesbit and J. Smith, "Data Cache Prefetching Using a Global History Buffer," in *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. IEEE, 2004, pp. 96–96.

[7] R. Cooksey, S. Jourdan, and D. Grunwald, "A stateless, content-directed data prefetching mechanism," *ACM SIGPLAN Notices*, vol. 37, no. 10, p. 279, Oct. 2002.

[8] J. Collins, S. S. Sair, B. Calder, and D. M. Tullsen, "Pointer Cache Assisted Prefetching," in *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, 2002, pp. 62–73.

[9] A. Roth and G. Sohi, "Effective jump-pointer prefetching for linked data structures," in *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*. IEEE Comput. Soc. Press, 1999, pp. 111–121.

[10] G. Plumbridge, J. Whitham, and N. Audsley, "Blueshell : A Platform for Rapid Prototyping of Multiprocessor NoCs and Accelerators." University of York, 2013, p. To Appear.

[11] Xilinx, "7 Series FPGAs Memory Interface Solutions," 2011.

[12] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 63–74, 2007.