Investigating Shared Memory Tree Prefetching within Multimedia NoC Architectures

Jamie Garside
Dept. of Computer Science
University of York, UK
jg599@york.ac.uk

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

Abstract— **This paper provides further evaluation of the proposed hardware prefetch unit for the Blueshell NoC. This utilises a separate shared memory tree (Bluetree) for connecting CPUs to external memory. The tree is supplemented with a Prefetch Unit next to external memory. Prefetching is carried out in a streaming manner, with prefetch distance being varied between 1 and 4. Whilst previous work has suggested that prefetching is an appropriate architectural technique within NoCs, enabling better system performance, this paper provides further behavioural insight – particularly the degree to which the bottleneck of external DDR (single port access) eventually dominates performance.**

Evaluation via traffic generators (hosted on Microblaze CPUs in the NoC) show improvements of over 100% for certain memory loads and prefetch distances. In all cases, prefetching is shown to have a beneficial effect upto the point at which the memory system is flooded by CPU requests. The evaluation is supported by an MP3 case study, which shows improvements of around 10% for upto 4 CPU cores – performance improvement falling as the number of CPUs increases (to 8 or 16) due to the memory system being flooded.

I. Introduction

A key challenge is to ensure that CPUs have sufficient bandwidth to external memory. However the relative performance gap between a CPU and its attached memory is continually widening [1]; the inclusion of multiple CPUs on a shared bus merely exacerbates the inevitable memory bottleneck [2]. Architectures such as Network-on-Chip (NoC) [3] offer different tradeoffs, reducing shared bus contention but introducing latencies due to memory accesses being routed across the NoC mesh, which may be contended. Whilst hardware prefetching has been widely used in conventional architectures to increase memory performance and reduce latencies [1], [4], its benefits have not been widely researched for NoC architectures.

Whilst commodity CPU architectures often include hardware prefetch to improve performance (eg. [5]), typical NoC architectures do not typically include hardware prefetch (eg. [6], [7], [8]). Indeed, a common assumption is that CPUs within the NoC either do not have cache memory, or that at run-time there are no cache misses instigating a off-chip memory transaction[1].

[1]That is by ensuring that all code and data needed by application threads / tasks running on a CPU are no larger than the cache size of the CPU. If cache misses are permitted, the NoC must be managed to avoid contention delays – eg. by using a TDM scheduled NoC to manage external memory accesses [9].

In [10], [11] the Blueshell NoC is introduced. This is a conventional Manhattan mesh NoC [3], incorporating a number of interconnected CPU tiles (see Figure 1). Blueshell utilises a separate shared memory tree (Bluetree) for connecting CPUs to external memory to reduce contention and bottlenecks. In [11] a Prefetch Unit is included next to external memory (see Figure 2). This paper investigates further the performance of *hardware prefetch for NoC architectures* within the context of the Blueshell NoC, which allows prefetch to CPU cache (which is dual-ported to avoid stalling the CPU).

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 describes the overall architecture (including NoC and memory hierarchy). Evaluation is given in section IV, including an MP3 decoder case study (section V) and conclusions drawn in section VI.

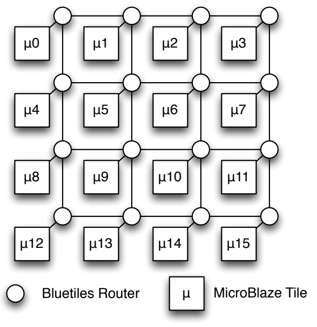


Fig. 1. Blueshell Network-on-Chip with 16 CPU Tiles

II. Background and Related Work

Prefetching is a used within CPU architectures [1] to fetch data and/or instructions across the memory hierarchy to be closer to the CPU. The goal is for the data / instructions to be available just before they are needed by the CPU, thereby improving performance. Within this paper, we confine consideration of prefetch between off-chip memory and CPU cache.

Hardware prefetching is implemented by monitoring memory accesses between CPU and cache, predicting the locations of future memory accesses, then issuing appropriate prefetch

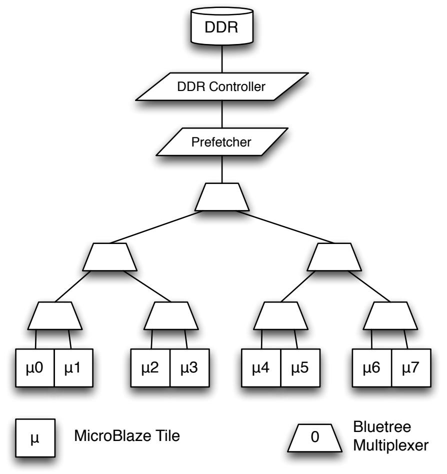


Fig. 2. Bluetree Shared Memory Tree for 8 CPU Tiles
Each CPU tile in the Bluetiles NoC (see figure 1 also connects directly (via cache) to the Bluetree shared memory (binary) tree to access external DDR memory, shown above. There is no interference between CPU to CPU messages across the NoC and CPU to memory transactions across the tree.

requests to memory to fetch those locations to cache. Stream prefetching [4] assumes that if memory blocks at addresses $n-2$, $n-1$ and n have been accessed in sequence, the CPU will soon require block $n+1$. 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, [12] developed a prefetcher which if blocks $n-2d$ and $n-d$ have been requested, assumes that the CPU will require blocks n and $n+d$ in the near future. However, for a large loop kernel with many loads, there must be a sufficiently large table in order to store all of this information. [13] shows that for program counter-based approaches such as this, a table of minimum 256 elements is required for adequate performance.

Data prefetch is more complex, as access patterns are more unpredictable. For every memory address causing a cache miss, Markov prefetching [14] stores which memory addresses came next in the reference stream and prefetches those addresses. Other approaches either examine data fetched from memory to find memory addresses (i.e. pointers to data in a linked data structure) [15], [16] or require the programmer to annotate their data structures with prefetch candidates [17].

Prefetchers 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 Overview

This paper assumes the Blueshell NoC framework [10], [11] – basic architecture is shown in Figure 1 and 2. Specific configuration of the framework for this paper includes:

- *Routers (Bluetiles)* (figure 1):
  Routers are 32-bit bi-directional with X-Y routing used (destination is contained in the first word). We note that the choice of routing policy does not impact upon the research presented in this paper, since this research focuses on the communication over Bluetree; Bluetiles is only used for simple synchronisation.
- *Shared Memory Tree (Bluetree)* (figure 2):
  2-to-1 multiplexors form a tree connecting CPUs to memory – CPUs are the leaves of the tree, memory being 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. Each multiplexor port allows 128 bits of data (corresponding to the cache line size); with single latches at all three ports (hence 2 cycles for data to traverse the multiplexor).
- *Prefetch Unit (PU)* (figure 2):
  Allows prefetch of external memory to the caches within the CPU tile. Prefetching is carried out in a streaming manner, with prefetch distance being varied between 1 and 4. See [11] for details of the PU.
- *CPU Tiles* [10], [11]:
  CPU tiles are built using the Microblaze CPU [18]. CPU configuration is 8kB split data and instruction caches, and a 8kB shared scratchpad used for fast local storage. The CPU accesses the cache via Microblaze LMB interfaces; cache misses being issued to external memory via Bluetree. The CPU tile contains custom cache control is configured to allow selective invalidation of cache lines and to record prefetch related data on a per cache line basis (the cache control unit also serves as the Microblaze's interrupt controller and provides a clock-cycle counter facility). Cache control is accessed via Microblaze Fast Simplex Links (FSL), utilising single-cycle FIFOs. Further details of the cache design are given in [10], [11].

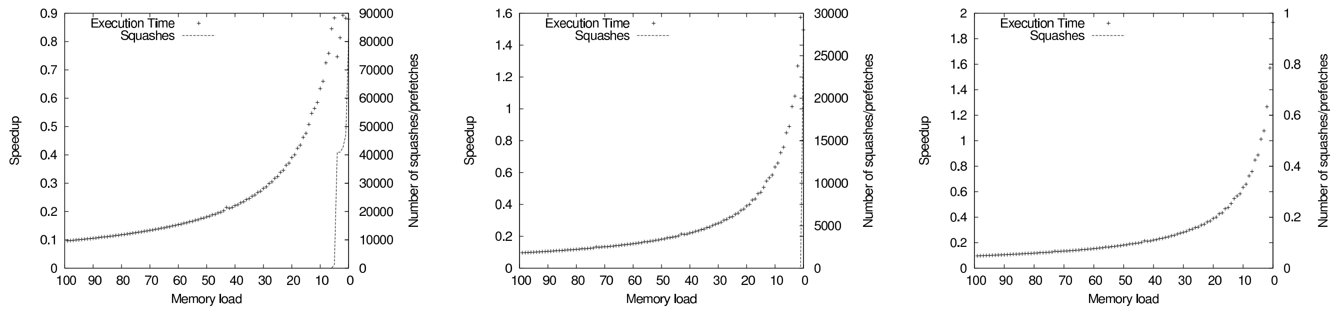
The Blueshell NoC framework is a collection of libraries written in Bluespec System Verilog [19]. A typical NoC is implemented with some extra tiles to aid bootstrapping, debug and obtaining experimental results [10]. Access to memory mapped peripherals is supported by attaching the Bluetiles network to an AXI [20] bus via an AXI bridge tile. Fully working designs have already been created for the Xilinx ML605, Xilinx VC707 and Digilent Atlys Spartan-6 board.

IV. Evaluation

Initial evaluation was performed by using traffic generators, as described in this section. In section V the results of an MP3 based case study are reported.

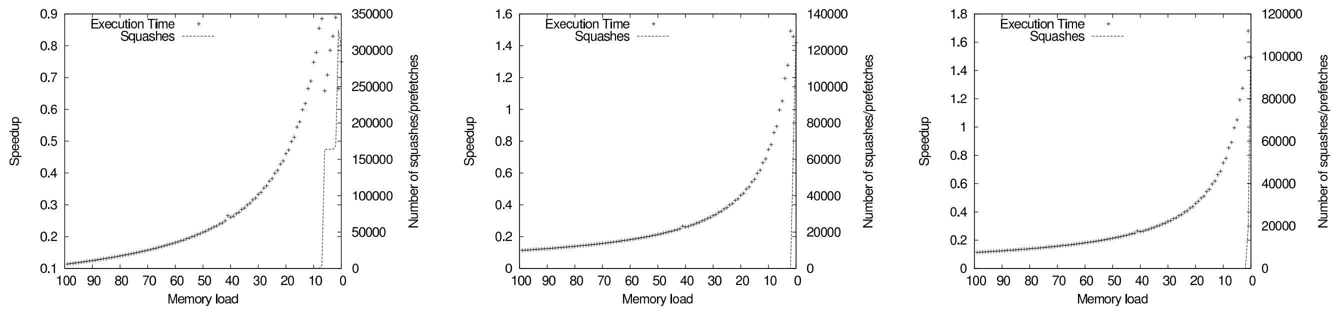
A. PU and NoC Implementation

The streaming Prefetch Unit (PU) was implemented within a 4x4 Bluetiles NoC (see section III), with each Microblaze CPU having a connection to the shared memory tree. The PU is at the root (see 2), hence a depth of 4 multiplexors between any



(a) 1 Lookahead (b) 2 Lookahead (c) 4 Lookahead

Fig. 3. Prefetcher behaviour for a single CPU.
Speedup of 0 represents no speedup; speedup of 1 represents 100% speedup. Memory load of 100 represents maximum delay (300 cycles); 0 represents minimum delay (ie. maximum memory load).



(a) 1 Lookahead (b) 2 Lookahead (c) 4 Lookahead

Fig. 4. Prefetcher behaviour for four CPUs.
Speedup of 0 represents no speedup; speedup of 1 represents 100% speedup. Memory load of 100 represents maximum delay (300 cycles); 0 represents minimum delay (ie. maximum memory load).

CPU and the PU, and a further multiplexor between PU and DDR (not illustrated in 2). The target for the implementation was a Xilinx Virtex-7 FPGA attached to DDR3-800 external memory, allowing CPUs within the NoC to run at 50MHz. When implemented, this prefetch unit consumes around 2400 slices and 7100 LUTs of the FPGA fabric for a 16-processor prefetcher. For the most part, the high slice utilisation is simply for storing data, such as the input/output FIFOs and squash buffer storage, which in total utilise around 4200 slice registers.

In terms of timing and latencies, we note [11]:

- *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 ≈15 clock cycles.
- *PU*: cost of crossing is 2 cycles.
- *DDR*: clocked at 100MHz, with a ≈ 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 ≈ 40 cycles. Hence total time is ≈ 60 cycles.

B. Traffic Generation and Experimental Parameters

The Prefetch Unit (PU) was initially evaluated via CPUs executing a traffic generator fetching from addresses N, $N+1$, $N+2$ etc in sequence, waiting in between accesses in order to simulate varying memory loads. Thus each fetch generates a load of a cache line (16 bytes) from memory. Note that a "burst" traffic generator (issuing a number of memory requests as fast as possible, then pausing) produces similar results and have been omitted for brevity.

Experimental parameters:

- *Number of CPUs*: 1, 4, 8 or 16 (this maintains a fully connected tree).
- *Prefetch distance*: 1, 2, or 4.
- *Delay between successive memory accesses by CPU*: 0-300 cycles in steps of 3 (as the loop body takes three cycles).

Note that 0 delay cycles places maximum load on the memory tree; at 300 delay cycles, the time spent in the delay loop will eventually dwarf the time spent accessing memory, with the overall speedup tending to zero.

C. Experimental Metrics

For each experiment the following were measured:

- *Speedup*: representing the speedup from enabling the prefetcher. A speedup of zero represents no speedup (i.e.

performance was identical with the prefetcher enabled and bypassed). A speedup of one represents a 100% speedup (i.e. performance with the prefetcher enabled was double that with the prefetcher bypassed). Negative speedups represent a performance penalty.

- *Memory Load*: equivalent to the delay which is inserted between memory accesses. A memory load of zero is no delay, which is effectively the maximum possible memory load for that system. A memory load of 300 is the maximum delay, which was the minimum memory load measured.
- *Number of Prefetches*: the number of cache lines which were prefetched at that memory load. In ideal cases, this should be constant for all parameters.
- *Number of Squashes*: the number of prefetches that were squashed (coalesced with an outstanding demand miss). In ideal cases, this should remain at zero.

D. Single CPU

The overall shape of the behaviour of the prefetcher with a single CPU can be seen in Figure 3. These graphs show the results for the 1, 2 and 4 lookahead variants. The graphs show that as the memory load increases, as do the potential performance gains from enabling a prefetcher. Note that the speedup tends to zero as the delay increases (and thus memory load decreases). If the speedup is measured only for the parts of the traffic generator that access memory (i.e. the delay loop is not profiled), the speedup is effectively constant.

Within Figure 3(a) we note the speedup flattening (at a load of about 5), and becoming noisy. This can be explained by the "squashes" line in the graph, that is, around this point, the prefetches are not being completed by the time that the traffic generator requests that data from memory. In real terms, this equates to around 16% of actual memory load for a single CPU. This figure is low as the routing time from the CPU to the memory controller and back again dominates the time taken to access DRAM from the DRAM controller.

As can be seen in figures 3(b) and 3(c), this effect can be alleviated by increasing the prefetch distance. This causes the prefetch for an address to be (effectively) issued earlier, hence there is a longer period of time between the prefetch being issued and required. This then reduces a likelihood of a squash, since the deadline for the prefetch to be completed is now further in the future.

E. Four CPUs

The graphs for 2 (not shown) and 4 CPUs (in Figure 4) are almost identical to those for a single CPU, albeit the "plateau" occurs at a slightly lower memory utilisation. Again, this plateau corresponds to an increasing number of "squashes", indicating that prefetches are not being issued fast enough.

Recall from section IV-D that this plateau occurred at around 16% of "real" memory utilisation for a single CPU. For memory utilisation $<16\%$ there is sufficient spare time where the memory is not being utilised, with memory requests originating from different CPUs easily interleaved. This effect can still be seen in Figure 4 – despite there being a number of squashes, there is sufficient time, either in the hold-off interval or the transit time for another task's access, to handle memory requests from other CPUs (so that the graph follows the same shape as shown in section IV-D).

F. Eight/Sixteen CPUs

The graphs for 8 (Figure 5) and 16 (Figure 6) CPUs show the effects of loading the memory tree so that there is insufficient time to support the interleaving mentioned in section IV-E. The first half of these graphs show an increase in performance (similar to other graphs for 1 or 4 CPUs), although the falloff for 8 and 16 CPUs is seen between loads 10 and 20, respectively. This peak is typically paired with a high number of "squashes", as before.

This effect damages performance since a prefetch is coalesced with its demand miss almost at the same time that the prefetch was issued. This means that performance degrades to the point where the prefetcher is effectively not doing anything, it may simply save one or two cycles at a maximum.

G. Further Observations – Tree Priority Encoding

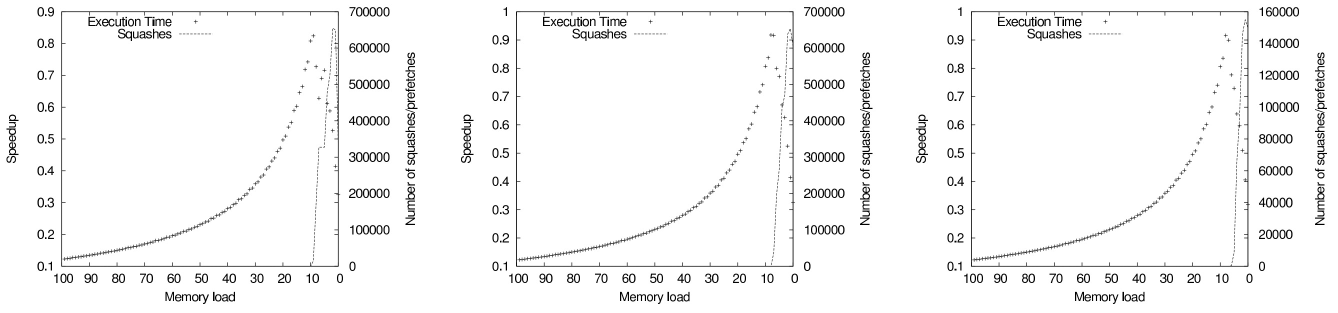
The Bluetree multiplexor has a fixed priority encoding – ie. requests from one port are always prioritised, assuming the requests arrive in the same clock cycle. This could cause a CPU to be locked out of memory. However, if the prioritised CPU makes a memory request, there is at least one free cycle whilst the upstream memory replies (to the issuing CPU), so the "non-priority" processor can issue a request.

There is a case where a processor emitting constant writes can lock the bus if a write is issued every cycle. This is unlikely for a couple of cases. First, given a cache lookup time, each instruction fetch is not single-cycle, thus a write cannot occur every cycle. Secondly, writing to sequential memory locations requires subsequent instructions to update the address and loop back. This can be avoided using loop unrolling, but this only alters the burst length; N addresses from N registers could be written before the values in these registers must be updated again, thus freeing the bus. Further experiments in the Bluetree platform which capture statistics such as "most utilised port" support this hypothesis.

H. Further Observations – Higher Workloads

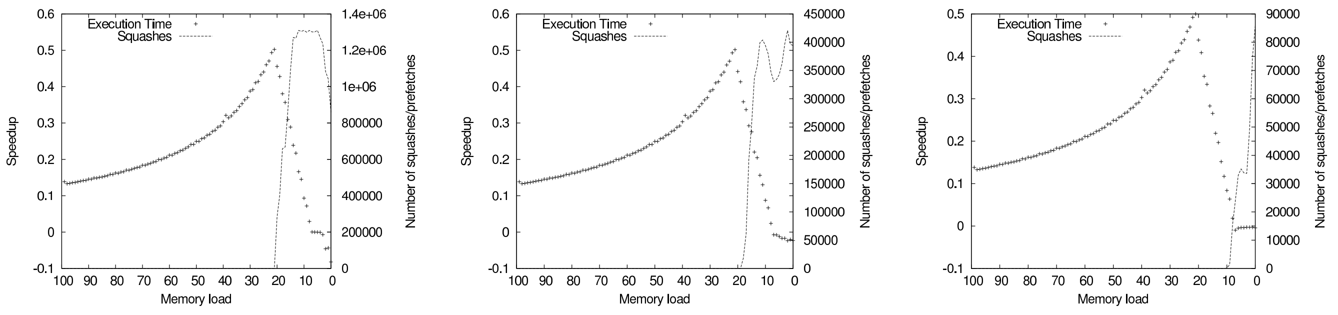
For higher workloads, as the memory load increases the number of prefetches begins to decline. This is due to the effective priority encoding between demand misses and prefetch accesses. At higher workloads, the memory becomes fully utilised with demand misses, with prefetches getting issued too late to have an effect on performance – ie. the prefetch is dispatched *after* the demand miss for the same line. Because it is late, it is not marked as a prefetched within the cache, hence there is no feedback from the cache when the line is used. In effect, on the next demand miss, the prefetcher must re-train, leading to a reduction in the number of prefetched lines, and as such worse performance.

A side effect of this can be seen at the right-hand side of figure 6(a). At high load prefetches are dispatched *too late* –



(a) 1 Lookahead (b) 2 Lookahead (c) 4 Lookahead

Fig. 5. Prefetcher behaviour for eight CPUs.
Speedup of 0 represents no speedup; speedup of 1 represents 100% speedup. Memory load of 100 represents maximum delay (300 cycles); 0 represents minimum delay (ie. maximum memory load).



(a) 1 Lookahead (b) 2 Lookahead (c) 4 Lookahead

Fig. 6. Prefetcher behaviour for sixteen CPUs.
Speedup of 0 represents no speedup; speedup of 1 represents 100% speedup. Memory load of 100 represents maximum delay (300 cycles); 0 represents minimum delay (ie. maximum memory load).

these prefetches have no effect, since the demand miss for that line has already happened, and thus the prefetch ties up the memory for a full cycle with an ineffective access, leading to a performance degredation as there is contention from other demand misses for that time slice.

V. Case Study

To demonstrate the performance of the PU on a real workload, the Helix fixed-point MP3 decoder [21] was ported to the Blueshell platform. This section describes the performance of the PU for this real workload.

A. Helix MP3 Decoder

The Helix decoder [21] has a lightweight footprint and minimal hardware requirements. It relies upon the memory allocation components of libc (which can be replaced if required) and a number of external math routines. Its computation is fixed point, removing the need for a hardware floating-point unit. When compiled, its memory requirements are around 40kB of data memory.

The Helix implementation was customised to parallelise the workload by placing the input data in main memory, then splitting the data between CPUs in a divide-and-conquer strategy. Of course, this strategy is not perfect for MP3. The specification of the codec allows "spare" data from each frame to contribute to a reservoir for use in subsequent frames [22]. By starting in the middle of the data, this reservioir will be empty and as such, a couple of frames will be discarded as it refills. For this reason, a real decoder should overlap the work done by each CPU so that this difference does not exist. This issue was ignored for a proof of concept, however as it has no effect on results.

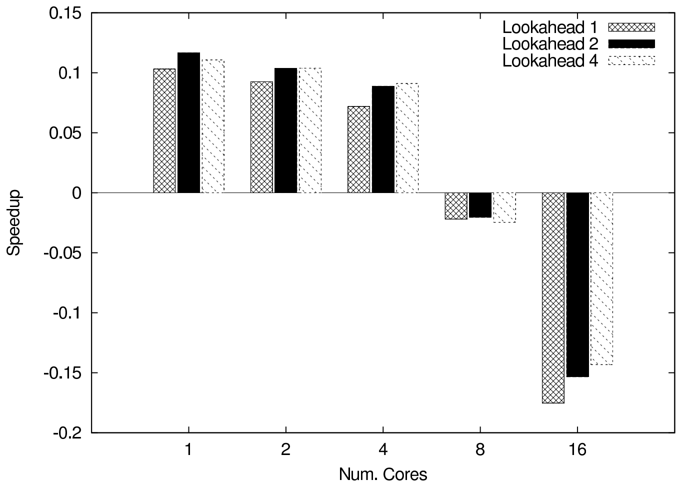


Fig. 7. A plot of the speedups from the Helix MP3 Decoder [21]
Speedup of 0 represents no speedup; speedup of 1 represents 100% speedup.

B. Results

Results are shown in figure 7. This shows the speedup, represented in the same way as in section IV and for identical system parameters (i.e. 1, 2, 4, 8 and 16 processors with a stride of 1, 2 and 4). The speedup is derived over the whole execution time of the decoder (including setup/teardown) on a 45-second 128kbps audio file.

The MP3 decoder is, in effect, a memory bound application; all CPUs must parse a section of the audio data in an incremental fashion, which is ideal for detection by a prefetcher. This also means that the workload is very sensitive to memory bottlenecks. For these reasons, prefetch for MP3 decoding should be effective for high memory utilisations.

The graphs (figure 7) for one, two and four CPUs show a good speedup of around 10%. In addition, the changing prefetch distance can improve performance. This is due to sections of the MP3 file being copied into internal buffers using the `memcpy` function, hence the memory utilisation in these sections is the highest it can be; `memcpy` can be implemented in around 4-5 machine instructions. The reason for the speedups from an increased stride are hence the same as in section IV. There are also other similar speedups throughout from rapid access to memory buffers.

The graphs for eight CPUs, however, show a very steep drop-off into performance degradation. The reason for this is very similar to the reasons outlined in IV-F. An MP3 decoder, of course, has a high number of reads from successive memory locations, implying a large number of cache misses and hence a large number of memory reads, but in addition has a large number of writes of both intermediate data and output data. This latency of intermediate buffers is normally alleviated using a cache, but since Blueshell's cache uses a write-through design, these must all be also committed to memory, causing even more memory load.

These reads and writes end up driving the effective memory utilisation to 100% for most memory-heavy regions, blocking prefetches from being correctly dispatched from the PU. This leads to the same case as outlined in section IV-F, where prefetches simply cannot be dispatched in time, and thus end up fetching data which has already been fetched by a demand miss, tying up the memory controller further and leading to a performance degredation.

VI. Conclusions and Future Work

Standard prefetch techniques can work on a multi-core system such as a NoC, masking some 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).

Evaluation via traffic generators (hosted on Microblaze CPUs in the NoC) show improvements of over 100% for certain memory loads and prefetch distances. In all cases, prefetching is shown to have a beneficial effect upto the point at which the memory system is flooded by CPU requests. The evaluation is supported by an MP3 case study, which shows improvements of around 10% for upto 4 CPU cores – performance improvement falling as the number of CPUs increases (to 8 or 16) due to the memory system being flooded (ie. the external DDR).

Acknowledgements

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

References

[1] J. Hennessy and D. Patterson, *Computer Architecture : A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2012.

[2] H. G. Lee, N. Chang, U. Y. Ogras, and R. Marculescu, "On-chip communication architecture exploration," *ACM Transactions on Design Automation of Electronic Systems*, vol. 12, no. 3, pp. 23–es, Aug. 2007.

[3] W. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pp. 684–689.

[4] 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.

[5] Intel, "Intel 64 and IA-32 Architectures Optimization Reference Manual," Intel Corporation, Tech. Rep. 248966-027, 2013.

[6] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost, "HERMES: an infrastructure for low area overhead packet-switching networks on chip," *Integration, the VLSI Journal*, vol. 38, no. 1, pp. 69–93, Oct. 2004.

[7] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Computing Surveys*, vol. 38, no. 1, pp. 1–55, 2006.

[8] A. B. Achballah and S. B. Saoud, "A survey of network-on-chip tools," *International Journal of Research and Reviews in Computer Science*, vol. 2, no. 2, pp. 554–560, 2011.

[9] R. Stefan, A. Molnos, and K. Goossens, "dAElite: A TDM NoC Supporting QoS, Multicast, and Fast Connection Set-up," *IEEE Transactions on Computers*, no. PrePrints, 2012.

[10] G. Plumbridge, J. Whitham, and N. C. Audsley, "Blueshell : A Platform for Rapid Prototyping of Multiprocessor NoCs and Accelerators," in *Proceedings HEART Workshop*, 2013.

[11] J. Garside and N. C. Audsley, "Prefetching Across a Shared Memory Tree within a Network-on-Chip Architecture," in *Proceedings International Symposium on System-on-Chip*, 2013.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Xilinx, "MicroBlaze Processor Reference Guide UG081 (v14.2)," Xilinx Inc., Tech. Rep., 2012.

[19] Bluespec Inc., "Bluespec System Verilog (BSV) URL:http://www.bluespec.com/products," 2013.

[20] Xilinx, "AXI Reference Guide UG761 (v14.3)," 2012.

[21] Realnetworks Inc, “The Helix MP3 Decoder,” 2013.
[22] S. Hacker, *MP3: The Definitive Guide*, 1st ed. O’Reilly, 2000, vol. 1.