WCET Preserving Hardware Prefetch for Many-Core Real-Time Systems

Jamie Garside
Department of Computer Science
University of York
York, United Kingdom
jamie.garside@york.ac.uk

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

ABSTRACT

There is an obvious bus bottleneck when multiple CPUs within a Many-Core architecture share the same physical off-chip memory (eg. DDR / DRAM). Worst-Case Execution Time (WCET) analysis of application tasks will inevitably include the effects of sharing the memory bus amongst CPUs; likewise average case execution times will include effects of individual memory accesses being slowed by interference with other memory requests from other CPUs. One approach for mitigating this is to use a hardware prefetch to move instructions and data from memory to the CPU cache before a cache miss instigates a memory request. However, in a real-time system, there is a trade-off between issuing prefetch requests to off-chip memory and hence reducing bandwidth available to serving CPU cache misses; and the gain in the fact that some CPU cache misses are avoided by the prefetch with the memory system seeing reduced memory requests.

In this paper we propose, analyse and show the implementation of a hardware prefetcher designed so that WCET of application tasks are not affected by the run-time behaviour of the prefetcher, i.e. it utilises spare time within the memory system to issue prefetch requests and forward them to the appropriate CPU. As well as not affecting WCET times, the prefetcher enables significant reduction in average case execution times of application tasks, showing the efficacy of the approach.

1. INTRODUCTION

To ascertain an estimate for the execution time of modern systems utilising shared resources, each component must be analysable and predictable. In the scope of shared memory, this implies two things. First, the maximum time taken for a memory request to be serviced must be known (e.g. through [3]), and secondly, the amount of blocking when attempting to access the shared memory by other tasks must be known and bounded.

Techniques do exist in order to attempt to ascertain these figures for standard COTS memory systems [20, 10], which typically attempt to ascertain how many memory requests can be made by a higher-priority task in a set period and hence the worst case blocking for a task at a given priority level. These techniques analyse the system as a whole and model the interactions between tasks, which soon becomes infeasible or pessimistic as the number of possible tasks in the system increases. This is a problem which can be solved using a composable analysis [13], which splits the available bandwidth to shared resources amongst the tasks which require access. Each task then only needs to be considered within its own partition, vastly simplifying the analysis.

Each of these techniques do have their problems. While the worst-case response time of memory can be known and bounded, this bound must assume maximal blocking by all other tasks, i.e. each other task must fully utilise its available memory bandwidth. This may not actually be the case for the full life cycle of a task, and hence there may be some "spare" bandwidth which can be used for other uses.

One useful technique for this can be prefetching. Prefetching attempts to speculatively issue requests for required memory contents *ahead of time*, such that they will arrive at the processor before required. This can mask all or part of the delay associated with accessing memory. This technique can further complicate system analysis since the traffic generated by the prefetcher is typically unpredictable, data fetched by the prefetcher may displace useful cache data, or may simply be useless information.

Given these problems, we present a novel prefetcher design, coupled with a memory arbitration scheme, which allows prefetches to be dispatched, whilst still providing a guarantee on the worst-case behaviour of a system. This then allows the average case execution time for a task to be improved, whilst not affecting the worst-case.

The remainder of this paper is structured as follows: Section 2 will cover related work in the fields of prefetching and memory arbitration. Section 3 will provide the theory of the worst-case prefetcher. Section 4 will provide an overview of the design of the system, while Section 5 will provide a timing analysis of the memory interconnect. Finally, Section 6 will provide an evaluation of the overall system performance, and Section 7 will draw conclusions from this work and out-

This work has received funding from the European Union's Seventh Framework Programme under grant agreements FP7-ICT-288008 and FP7-ICT-318763

RTNS 2014 , October 08 - 10 2014, Versailles, France

ACM 978-1-4503-2727-5/14/10 $15.00.
http://dx.doi.org/10.1145/2659787.2659824.

line future work.

2. BACKGROUND & RELATED WORK

2.1 Prefetching

Stream prefetching [17] is a technique that makes the assumption that if memory addresses A, $A+1$ and $A+2$ have been accessed, then the data at address $A+3$ will likely be required in the near future. This is typically implemented using a lookup table keyed on the address of the last miss. On a cache miss, the miss address can be looked up in this table and if it exists, the record updated and a prefetch optionally dispatched. If there is no corresponding row, then a new row is added, replacing an existing row through the means of LRU, Round Robin, or a similar replacement scheme.

Stream prefetching can only detect serial streams with a known, fixed stride. *Stride* prefetching [4] can detect data streams, as before, along with the stride between them. In this case, if memory addresses A, $A+d$ and $A+2d$ have been accessed, then the data at address $A+3d$ will be required in the near future, for a constant *distance* d.

The implementation of such a prefetcher is similar to that of a stream prefetcher, although the lookup table is instead keyed on the address of the load instruction in the program code, rather than the address loaded. This then allows for fast detection of a stream with a given stride if there are a sequence of accesses with the same stride. This approach does have its problems though; it is unsuitable for code prefetch, since there is no program counter for a code load, and [12] show that for program-counter based approaches such as this, a table of around 256 entries is required to see a significant performance gain, and thus the prefetcher has a large hardware footprint.

Other prefetching such as Markov prefetching [16] can fetch a series of data without a predictable access pattern. In this case, the prefetcher stores each miss address, along with a set of the addresses accessed after said miss address, and the probability of those addresses being accessed. This can give good performance gains for unpredictable data streams (i.e. iterating over a dynamically-allocated linked list), although requires a huge table to store the required data. An improvement on this approach instead stores the deltas between memory addresses [19], such that if the difference the memory addresses of accesses A and B is d, then the next address will be $B+d'$ with a probability p', or could be $B+d''$ with a probability p''.

Other approaches to fetching unpredictable streams involve examining the data returned from memory for potential memory addresses (e.g. checking if a returned piece of data lies in the range of the heap or data sections, therefore is likely to be a memory address) such as [9, 8], and other approaches require the programmer to encode hints into the program to be used by the prefetcher, such as [24].

Other approaches such as Feedback Directed Prefetching [25] attempt to tune the parameters of the prefetcher at run-time based on the current behaviour of the system. As an example, if all prefetches are arriving too late, the prefetch distance can be increased in order to allow more time for a prefetch to be serviced and thus reduce the likelihood of it being too late. If a prefetcher is currently operating with high accuracy, the prefetcher can increase the prefetch degree in order to fetch more useful data at once, utilising the fast sequential speed of DRAM. Adaptive Stream Detection [15] builds a histogram of the lengths of data streams within a program. Using this information, the prefetcher can dynamically decide the prefetch degree at run-time.

2.2 Memory Interconnect

Many systems are starting to incorporate separate networks for communication traffic and memory traffic. The Tilera TilePRO [2] processor, for example, separates each communication type onto its own network. One rationale behind this is to prevent memory and I/O traffic, which must reach the memory and I/O controllers at the edge of the device, from interfering with inter-process communications which communicate within the device. This leads to a system with a physically separate communication network, memory network, I/O network and cache coherence network.

MEDEA [26] is another network-on-chip system which individually caters for the requirements of a core's inter-process communication and its memory access requirements. In this architecture, the memory I/O port of the processor and a programmable I/O port for communication are multiplexed onto the interconnect using a fair arbitration scheme. In addition, this architecture utilises deflection-routing, which is that every packet *must* be forwarded somewhere on every cycle. This can suffer from livelock and does not make any guarantees on the latency of messages.

The Bluetree [21] network-on-chip completely separates communication traffic and memory traffic in much the same way as the Tilera architecture, although utilises two completely separate interconnects in order to achieve this. This utilises a standard mesh network for communications, coupled with a tree-based interconnect optimised for connecting many processors to a single memory controller.

2.3 Timing Analysis

Two of the major approaches for ascertaining a worst-case execution time for a given task are measurement-based approaches (such as high watermark) and static code analysis. Tools such as Rapitime [22] and pWCET [5] attempt to ascertain the worst-case execution time through measurement of blocks of code in the system. This leads to accurate measurement of the charactisterics of the platform, rather than a model of the platform, although it is difficult to verify that the worst-case path has been taken and hence, the worst-case execution time estimate may be optimistic. Additionally, in multi-core systems, it is difficult to ensure that the task has received the maximum amount of blocking possible from other cores in the system.

Static techniques such as aiT [14], SWEET [11] and OTAWA [6] are static analysis based approaches. From a model defining the timing characteristics of a given system, this approach calculates the worst-case execution time for basic blocks by summing the worst-case execution time of each instruction or group of instructions. From this information, a worst-case path can be built and thus the worst-case execution time calculated. This is an approach which is more likely to be able to ascertain the worst-case path for a system, but typically requires the programmer to add annotations to bound certain behaviour of the task, for example, the iteration count for a loop and the potential values for indirect memory accesses.

3. PREFETCHING

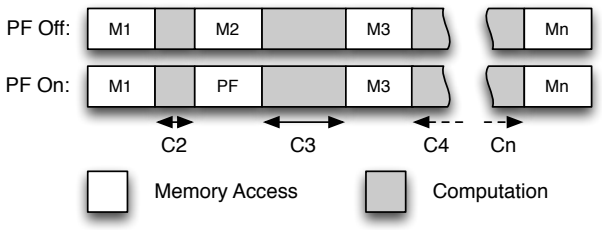


Figure 1: Example of a Memory Access Stream

As noted in Section 1, utilising prefetch can both harm and improve performance. In the ideal case, all prefetches are required by the CPU and arrive in a timely fashion, thus improving performance since much of the memory latency has been removed from the program flow. In the worst case however, none of the prefetches are required by the CPU. In this case, cache pollution will arise, displacing useful memory blocks and thus causing extra latency to re-fetch those useful blocks, and dispatching the useless reads to the memory controller will tie up the memory controller for a period of time.

Stemming from these limitations, we present a novel approach to prefetching which can utilise the worst-case analysis of a system in order to allow safe prefetching without harming the worst-case execution time of the system.

3.1 Using Hit Feedback

The first method of allowing prefetching without harming the worst-case execution time of a system is to exploit the prefetch hit feedback. This hit feedback is used by the processor to notify the prefetcher that prefetched data was, in fact, useful, which typically then causes the prefetcher to prefetch the next datum. This feedback can then be exploited in order to allow a prefetcher to operate in a safe manner.

A task can be modelled, from the perspective of the memory subsystem, as a stream of memory accesses $m_1, m_2, ..., m_n$. These memory accesses are then separated by an amount of computation consuming C cycles, forming a set of tuples $(m_1, C_1), (m_2, C_2), ..., (m_n, C_n)$. A graphical representation of this form can be seen in Figure 1. The time taken for each of these accesses can be bounded, given a known memory response time, then the task analysed in order to ascertain an execution time of the task.

Assuming that a prior prefetch has already fetched a memory address m_{pf} ahead of time, the fetch for m_{pf} will be removed from the memory access stream. Rather than removing this access from the stream, a different access can be dispatched when the access for m_{pf} would previously have been dispatched. This retains the previous behaviour of the memory stream, but allows a prefetch to be dispatched. A graphical representation of this can be seen in the second half of Figure 1; here, the access for m_2 has been prefetched ahead of time, and hence replaced with a prefetch.

This approach makes two assumptions: firstly, the access time to a memory location must be uniform, which can be achieved using a closed-page memory access policy (that is, a precharge is issued after access, rather than speculatively leaving the bank open for further accesses) [23]. Secondly, this currently assumes a single task per processor with no preemption. This approach can scale to multiple tasks with preemption; any memory fetches associated with cache-related preemption delays will form part of the reference stream, as normal, and the prefetcher need not measure the computation time between memory accesses C_x, instead the CPU can notify the prefetcher when a memory line would be fetched; the time C_x is merely an abstraction to demonstrate the prefetcher's concept.

3.2 Initial Prefetch

The theory in Section 3.1 assumes that a prefetch has already been delivered to a given processor. Delivering this initial prefetch can be performed in one of two ways, depending upon the characteristics of the target system:

3.2.1 Explicit Reservation

The prefetcher can be explicitly allocated a memory bandwidth budget within the chosen arbitration scheme. This allows for simpler analysis of the prefetcher's behaviour, since the amount of interference that the prefetcher can cause to other tasks is bounded and fixed. This does have problems inherent to other arbitration schemes though, such as that the prefetcher may not utilise all of its bandwidth budget, consuming resources which could be allocated to other tasks.

The bandwidth reserved to the prefetcher can then be used in the standard worst-case analysis. This allows for guarantees to be given to the prefetcher, at the cost of worst-case system performance.

3.2.2 Slack Stealing

The prefetcher can utilise any "spare" time within the system. As noted in Section 1, many arbitration schemes assign a memory quota to a task statically, which a task may not saturate during its whole life-cycle. In cases like this, the prefetch can "steal" this free time that the other tasks are not using. This is a more dynamic approach, although will make the behaviour of the prefetch more difficult to guarantee, without harming the worst-case analysis of the system.

4. SYSTEM DESIGN

In order to support worst-case aware prefetching in hardware, a hardware implementation of a prefetcher implementing the concepts outlined in Section 3 has been implemented using Bluespec System Verilog and successfully implemented on a 16-core system utilising the Blueshell Network-on-Chip [21]. In addition, the Bluetree multiplexers have been extended to include an arbitration scheme suitable for ensuring bandwidth guarantees to shared memory. This hardware implementation is clocked at 100MHz, connected to a DDR3 memory clocked at 200MHz.

4.1 Interconnect Design

A standard Bluetree multiplexer is simply a 2-into-1 multiplexer with an configurable static priority favouring one of the inputs. There is no rate limiting; a requestor which requests every cycle will simply block any lower priority requestors from being able to access memory, leading to a lack of composability in the platform.

In order to make the Bluetree system composable and timing predictable, an arbitration scheme has been built into each multiplexer. This scheme defines a *blocking factor* m, defined to be the period over which a single low-priority packet can take priority over a high-priority packet. In more logical terms, this scheme implies that a low-priority packet can be blocked by *at most* $m-1$ high-priority packets before

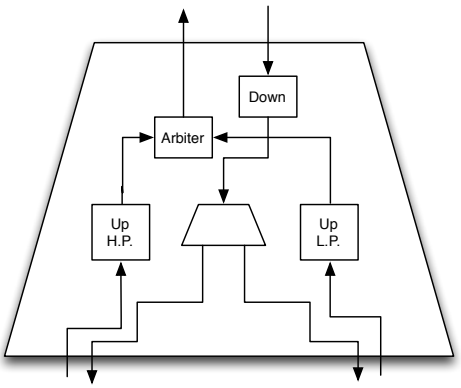


Figure 2: Internals of a Bluetree Multiplexer

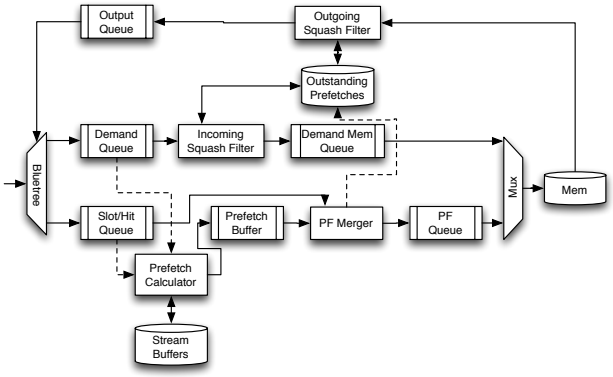


Figure 3: Block diagram of the prefetcher

being allowed service. Conversely, a high-priority packet can be blocked by a single low-priority packet in the worst case. In the case of the 2-to-1 multiplexers used, the "high-priority" side is the left-hand side of the multiplexer, with the "low-priority" side being the right-hand side, although this behaviour is configurable.

The multiplexer comprises an input buffer for each side, which are then multiplexed onto an output wire using an arbiter as described above. These input buffers support simultaneous read and write, that is, an input packet can be written into them on the same cycle that one is consumed by the arbiter. The downwards path is a simple demultiplexer, fed by a single input register. The downwards path is defined to be non-blocking, and as such, packets must be handled as soon as they become available. For this reason, no arbitration is required on the downwards path. A block diagram of this can be seen in Figure 2.

Given this arbitration scheme, timing guarantees can be ascertained for memory transactions, since the amount of blocking is known and bounded. A full analysis of this blocking system can be found in Section 5.

This distributed arbitration approach was chosen to attempt to work around the scalability problems inherent in monolithic arbiters. Typical arbiters demultiplex the memory stream into a number of virtual channels, then perform arbitration over these virtual channels and multiplex the output back onto the connection to memory. As the required number of virtual channels increase, so does the complexity of these multiplexers/demultiplexers, and thus the size increases and the possible maximum frequency decreases [7].

In order to support prefetch without harming the worst case, a mechanism to notify the prefetcher of available system slack is required, and that the prefetcher can therefore dispatch a prefetch.

It can be noted that if a low-priority packet is *not* blocked by any high priority packets, then it is effectively being dispatched as a work-conserving access. Similarly, if m high-priority packets can be dispatched without being blocked by a low-priority packet, then the mth packet is being dispatched in work-conserving mode. In these cases, it is possible to dispatch a "prefetch slot" instead. These are empty packets which take the place of an access that would have been dispatched if the multiplexer was fully loaded. These "prefetch slots" can then be filled in by the prefetcher.

In addition to dispatching prefetch slots, the multiplexers have also been designed to include a "squash detector". This works on an identical concept to the squash detector within the prefetcher; if there is a prefetch travelling down through a multiplexer towards a processor, and the multiplexer also has a read request for the same memory address, then the prefetch is transformed into a read response to fulfill the outstanding read. In addition, this will cause a prefetch hit request to be relayed up to the prefetcher. This allows prefetches to be coaleasced with their reads anywhere in the tree, reducing the number of redundant prefetches and thus reducing the amount of traffic on the tree.

4.2 Prefetcher Design

The implemented prefetcher is a simple stream prefetcher, designed to be placed in-between the memory tree for clients and main memory. Its location at the top of the tree allows it to snoop all memory accesses, and ascertain the state of main memory (i.e. how much bandwidth is available to it). The prefetcher is designed to operate using prefetch "slots" as described in Section 4.1. These are empty packets sent from the memory tree to the prefetcher which can be filled in. This allows the tree to be able to notify the prefetcher of slack time, as described in 3.2, which can then be used for a prefetch without harming the overall WCET.

In addition, from Section 3.1, a prefetch hit can then cause another prefetch to be dispatched without harming the WCET of the system, since the hit request takes the place of a "standard" memory request. This allows both prefetch hits and prefetch slots to become placeholders for standard prefetches.

If there are no available prefetch slots, then the prefetch slot is abandoned without being filled, and hence only incurs a single cycle penalty.

The architecture of the prefetcher is shown in Figure 3 and described below.

- *Demand Queue:* Packets are demultiplexed into one of two queues depending upon their type. Demand misses (i.e. cache misses) are split from prefetch hit notifications and slots, as the two are processed differently.
- *Hit/Slot Queue:* Stores prefetch hits and prefetch "slots" from the tree. Since a prefetch hit can be treated as a slot, it enters the same queue.
- *Prefetch Calculator:* takes information from the input queues, and using the stream buffers, decides whether to dispatch a prefetch.

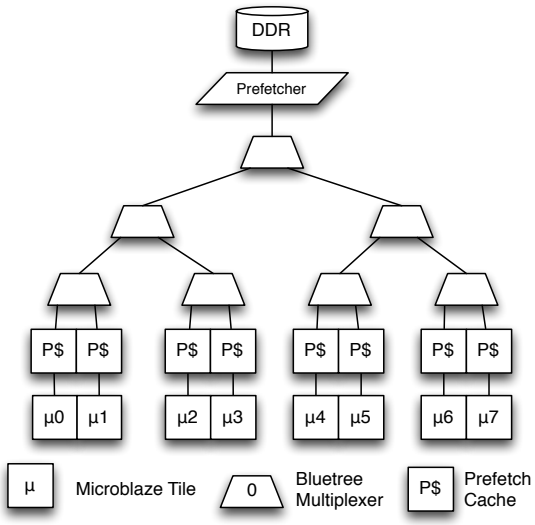


Figure 4: Block Diagram of Bluetree for Eight Processors

- *Stream Buffers* store the currently existing stream data. This stores a table for each CPU, storing the last addressed fetched, and if the stream is valid.
- *Demand Mem Queue:* Stores memory requests which have not been coaleasced with a prefetch and therefore need to be dispatched to memory.
- *Prefetch Buffer:* Stores outstanding prefetches, ready to be inserted into a prefetch slot.
- *PF Merger:* Takes slots from the *Hit/Slot queue* and merges them with a prefetch. It also then inserts an entry into the *dispatched prefetches* table, so that the prefetch can be coaleasced with a demand miss for the same line.
- *PF Queue:* Finally stores merged prefetches, ready to be dispatched to memory.
- *Mux:* multiplexes the demand miss queue and the prefetch request queue onto memory.
- *Incoming/Outgoing Squash Filters* are used in conjunction with the *dispatched prefetches* table, and are used to coaleasce prefetches with their demand misses. When a prefetch is dispatched, the prefetcher adds a row to this table to denote the outstanding prefetch. The incoming squash filter checks this table, and if a record exists for the current request, will discard the request and mark the table row as "squashed". The outgoing squash filter checks the prefetch's address against the relevant table row, and if the row is marked as "squashed", will return the prefetch as a demand read rather than a prefetch.

4.3 System Integration

The prefetcher described in Section 4.2 has been coupled to a 16-core Microblaze system, with the Microblaze processors connected to memory via the Bluetree multiplexers described in Section 4.1. Each multiplexer is a 2-to-1 multiplexer, hence the integrated system requires 15 multiplexers. In addition, each multiplexer has been configured with $m = 4$, hence a low-priority can be blocked by three high-priority packets before being allowed service. This number was chosen simply to explore the behaviour of the prefetcher on a non-uniform system.

In addition, as noted in Section 3, prefetching directly into the cache of the target processor may cause the worst-case execution time of a task to *increase*. For this reason, prefetches are instead delivered into a next-level cache. This "prefetch cache" need only be small, since the granularity of prefetches is so small, and need only be simple, hence is implemented as a simple direct-mapped cache.

Since each processor only has eight stream buffers, each of which can be used to fetch four words from memory, it makes sense that this cache need only be of size 32 bytes. In reality, this has been implemented as a 512 byte cache in order to alleviate problems arising from cache locality (i.e. fetches from different prefetch streams occupying the same cache lines). An example of this system design for an eight-processor system can be found in Figure 4.

5. WORST-CASE BEHAVIOUR OF THE BLUETREE MEMORY TREE

For the analysis of a Bluetree multiplexer, we assume the same design decisions as used in Section 4, that is, that there is a single input register on each channel, which supports simultaneous read and write. In addition, the downwards path can never block. We also assume a blocking factor of m, such that a low-priority packet can be blocked by at most $m - 1$ high-priority packets, and conversely, a high-priority packet can only be blocked by a single low-priority packet. Finally, we assume that the worst-case response time of a memory transaction is bounded from the time at which it enters the memory controller (i.e. is accepted from the top of the tree) and is represented using t_{mem}.

We also assume that the clients in the system are multi-issue. Whilst simple CPUs often can only have one outstanding memory request at any time (i.e. the CPU is effectively stalled until the memory request has been serviced), commercial CPUs can have a number of memory requests (specifically a maximum number of writes and a maximum number of reads) outstanding at any time. For example a quad-core Intel Xenon X3520 (Nehalem) [18] allows a total of 32 outstanding read and 16 write requests from all cores; whilst a single core can generate upto 10 concurrent read requests. Therefore within this paper, when analysing the worst-case performance of the memory system, we assume a memory request can be issue by each CPU on each clock cycle – which is pessimistic when compared to actual CPUs (including Nehalem), but assuming only one outstanding request at a time would be optimistic and lead to erroneous analysis and system behaviour.

First, we present an analysis of a single multiplexer in isolation, then extend this to provide a model to ascertain the amount of blocking that can occur on a tree of connected multiplexers.

5.1 Single Multiplexer

The behaviour of a single multiplexer can be easily predicted for a given blocking factor. The number of times that a packet may be blocked, B can be ascertained using Equation (1).

$$B_{up} = \begin{cases} 1 & \text{High Priority path} \\ m-1 & \text{Low Priority Path} \end{cases} \quad (1)$$

From this, the worst-case blocking can be ascertained us-

ing the following equation:

$$t_{up} = (B_{up} + 1) \times t_{mem} \tag{2}$$

This situation will occur when a packet experiences its worst-case blocking at the multiplexer as described in Equation (1). In addition to this, since we assume that a multiplexer's input buffers support simultaneous read and write, a packet must wait for the packet which preceeded it to be fully serviced, hence the additional t_{mem} delay.

After being serviced, a request must then travel back down the tree again. Since we make the assumption that a downwards packet cannot be blocked, and that crossing a multiplexer will take a single cycle, this delay, t_{down} is simply 1. The delay for a memory request crossing a single multiplexer is then as in Equation (3).

$$t_{mux} = t_{up} + t_{mem} + t_{down} \tag{3}$$

That is, the blocking experienced on the upwards path, plus the time taken to service the memory request and a single cycle to be returned back to the requestor.

5.2 Multiple Multiplexers

As a set of multiplexers are connected into a tree, the analysis is complicated by the fact that multiplexers can now block full subtrees. Assuming that the subtree is experiencing worst-case conditions, that is, that all of the buffers in the sub-tree are already full, no packets will be relayed within the subtree. Since no packets are relayed within the subtree, none of the blocking counters will be updated; the subtree is effectively stalled.

The worst-case blocking can occur for a packet when all multiplexers are blocking the path to memory. In this case, the packet will experience a large amount of blocking before being able to be relayed beyond the first level of multiplexers. As it progresses up the tree, it will experience further blocking as the multiplexers it still has to cross relay more packets.

This blocking significantly complicates the analysis in order to ascertain a tight bound. In order to perform this analysis then, we define three piecewise functions. We also initially assume that each block is for a single cycle, in order to simplify the definition of C_l. This can later be expanded to support full memory transactions by simply multiplying through by t_{mem}. Finally, we define a *priority path*, P. This encodes the path from a processor to the root of the tree, and the sides of the multiplexers the processor is connected to (L/R). For example, $P = \{L\}$ defines a system with a single multiplexer, where the processor is connected to the left-hand side of the multiplexer. $P = \{L, R, R\}$ defines a system with three levels of multiplexers, where processor is connected to the right-hand side of the bottom multiplexer, which in turn is connected to the right-hand side of the next multiplexer, which finally then connects to the left-hand side the of root multiplexer. We finally then define $P(l)$, which is the connection side for level $l > 0$.

- $C_l^P(t)$: The current internal cycle for the multiplexer at level l in relation to the current global time t, given a priority path P.
- $B_l^P(t)$: Specifies whether the input to the multiplexer at level l given a prioriry path P will be blocked in cycle t.
- $\hat{B}_l^P(t)$: Specifies whether a multiplexer at level l will be blocked by those multiplexers above it at time t, given a priority path P.

The definitions of these functions are provided below:

$$C_l^P(t) = \begin{cases} 0 & t = 0 \\ C_l^P(t-1) & \hat{B}_l^P(t-1) \\ C_l^P(t-1) + 1 & !\hat{B}_l^P(t-1) \end{cases} \tag{4}$$

$$B_l^P(t) = \begin{cases} True & C_l^P(t) \bmod m = 0 \wedge P(l) = L \\ False & C_l^P(t) \bmod m = 0 \wedge P(l) = R \\ False & C_l^P(t) \bmod m \neq 0 \wedge P(l) = L \\ True & C_l^P(t) \bmod m \neq 0 \wedge P(l) = R \end{cases} \tag{5}$$

$$\hat{B}_l^P(t) = \begin{cases} False & l = 1 \\ True & B_{l-1}^P(t) \\ \hat{B}_{l-1}^P(t) & !B_{l-1}^P(t) \end{cases} \tag{6}$$

That is, the multiplexer cycle will only advance if the multiplexer was not blocked by anything above it. Due to the definition of $\hat{B}^P(t)$, the top level multiplexer (at $l = 1$) can never block. Additionally, the high-priority path will be blocked if it has admitted $m - 1$ packets already, due to the mod function.

The worst-case blocking can then be ascertained using the $\hat{B}^P(t)$ function. We define the $L_l^P(t)$ function, which defines how far a packet, starting at level l has travelled at time t on a priority path P.

$$L_l(t) = \begin{cases} l & t = 0 \\ L_l(t-1) & B_l^{HP}(t) \\ L_l(t-1) - 1 & !B_l^{HP}(t) \end{cases} \tag{7}$$

That is, a packet begins at level l, and can only progress upwards if no multiplexer at a level above the current level blocks it. The worst-case blocking time is then

$$t_{mem} \times \min_{t=0}^{\infty} t : L_l(t) = 0 \tag{8}$$

The worst-case blocking is calculated by finding the t for which a packet has reached the top of the tree, and is then multiplied by the worst-case memory delay. Example figures for the number of blocks while transiting the tree for varying blocking factors can be found in Table 1. Given that the worst-case blocking has now been calculated, the worst-case response time of memory can simply be ascertained using the result of Equations (8) and that of the downwards path, which is simply the number of levels l. The worst case is thus:

$$(t_{mem} \times \min_{t=0}^{\infty} t : L_l(t) = 0) + l \tag{9}$$

That is, the worst case blocking on the tree, plus the time to clear the packet which was dispatched when a given packet was admitted, plus the processing for a given packet and finally the path back down the tree.

6. EVALUATION

The combination of the prefetcher with the memory tree has been built and evaluated using two different benchmark

Proc. Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
$m = 2$	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
$m = 3$	15	18	24	32	29	33	47	60	30	36	48	63	57	66	93	120
$m = 4$	11	16	26	39	28	44	71	114	32	48	76	116	84	132	212	340
$m = 5$	10	17	27	50	33	58	102	195	40	65	105	200	130	230	405	780

Table 1: Worst-Case blocking across a 16-core tree, measured in number of blocks.

strategies. Firstly, synthetic traffic generators were used. These issue a memory request to adjacent cache lines, with a configurable delay between each memory access. This delay is configurable between 0 and 1500 cycles, in increments of 3 cycles. This granularity is due to the fact that the delay loop takes three cycles to execute. This loop will repeat over 1024 cache lines, then report the execution time. This is repeated 20 times and the average taken with the prefetcher disabled, then another 20 times with the prefetcher enabled.

Secondly, a subset of the TACLeBench suite of benchmarks are used [1]. These are a suite of benchmarks deesigned to benchmark WCET analysis tools, but chosen since there are no external library dependencies, and no external stimulus (or randomness) is used. This is beneficial for these experiments, since the timing behaviour of the benchmarks *should* be identical on each run, and hence the effect of the prefetcher on the benchmark can be demonstrated.

In addition, two hardware configurations are used. First, a set of synthetic systems have been built, each of which contains a single processor and fifteen hardware traffic generators to simulate the system in "full load" conditions. These traffic generators simply request from address zero on every cycle for which their output queue is not full, and hence will cause all buffers on their path to memory to be always full. Additionally, since they only fetch from address zero, no prefetches will ever be dispatched for the traffic generators. Each different system contains the processor in a different location, thus the timing behaviour for processors in different locations on a fully loaded tree can be ascertained, given the different blocking factors as seen in Table 1.

Since the benchmarks used are single path and do not take any input from the outside world, this hardware platform can be used as a measurement-based approach in order to evaluate the worst-case behaviour of the system. In addition, by enabling the prefetcher in these systems, it is possible to demonstrate that the prefetcher does not cause a detriment to performance in worst-case conditions.

A system has also been built using sixteen Microblaze processors. This is a system which can show the timing behaviour on a system which only contains real-world traffic patterns. In this case, the tree should not be fully loaded at any point.

6.1 Traffic Generators

Figure 5 shows three example runs of the traffic generators for the processors at indices 1, 6 and 15, in the "full load" configuration with the prefetcher enabled and disabled. These three graphs show how the performance of the prefetcher changes as the amount of available bandwidth changes.

Figure 5a shows the behaviour of the traffic generator to a system with a processor at index 1 (i.e. a priority path of $\{HP, HP, HP, LP\}$ from the root of the tree). For the "prefetch off" line, a series of "steps" can be seen. These are due to the blocking introduced at the lowest level of the tree, and thus all accesses within a given period of each other will experience an identical delay.

The result with the prefetcher enabled can be seen to be a smoother line. This is because, up until a delay factor of around 100, a prefetch can be successfully dispatched and delivered into the processor's target cache before it is required. In this case, the blocking while waiting for a memory access to complete is effectively zero, since all memory accesses have been completed ahead of time, and thus the execution time is just the time taken to execute the delay loop.

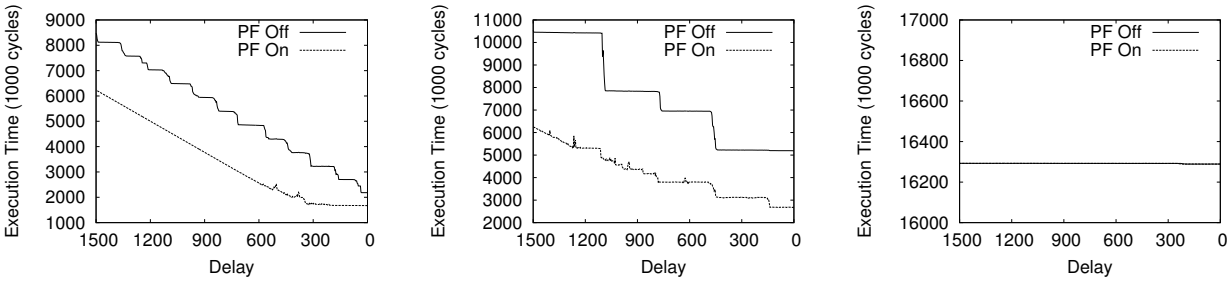
For a blocking factor *under* 100 however, the results begin to level off. This is because prefetches start to be coaleased with their demand misses, and hence "squashes" occur.

Figure 5b shows the same behaviour for index 6 (i.e. $\{HP, LP, LP, HP\}$). Like the processor at index 1, this shows an improvement which begins to level off. Of note is the fact that the "prefetch on" line begins to show "stepping" too. This is for a similar reason to why the "prefetch off" line shows steps, although is to do with the amount of blocking that hit feedback and prefetch slots will experience, since these messages are not handled specially. Figure 5c then shows the case where there is not enough available bandwidth to be able to dispatch a prefetch. In this case, no slots are generated, thus no prefetches can be dispatched and no prefetch hits can be generated. It can further be noted that even in these systems which do not have much memory bandwidth for prefetching, that the prefetcher can either improve performance or achieve identical performance, but never cause a task to perform worse.

Figure 5b shows some spikes in the prefetch on stream (at around 430 and 320). Recall from Section 4 that the prefetcher and multiplexers are able to coaleasce prefetches and demand accesses for the same addresses, preventing demand accesses from being issued immediately after their prefetch. It is not possible in all cases to coaleasce these accesses (e.g. when a demand access is relayed up from one multiplexer in the same cycle that the prefetch enters the same multiplexer to be relayed down). In this case, not all prefetches can be coaleasced, causing a slight performance detriment.

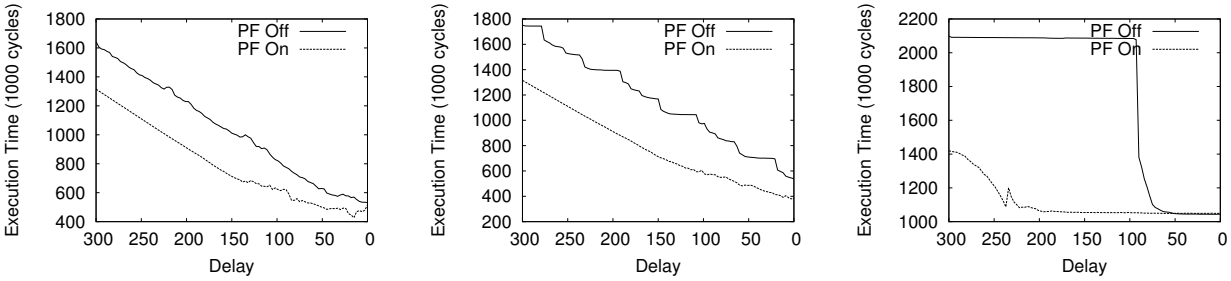
Figure 6 shows similar traces for the traffic generators, but on the system with sixteen processors, each running the traffic generator with identical settings. In these systems, the tasks will experience less blocking throughout the tree, and many more prefetch slots can be dispatched. Again, Figures 6a, 6b and 6c show the behaviour for the processors at indices 1, 6 and 15, respectively.

The stepping effect can still be seen in Figure 6b, but to a lesser degree. This is because there is still contention in the system and thus there will still be some blocking evident. The smaller blocking therefore still exists due to blocking further up the tree. This effect is still extremely evident in Figure 5c, which will experience the most blocking out of all of the processors. This graph also shows the spikes seen in



(a) Index 1 (b) Index 6 (c) Index 15

Figure 5: Plots of prefetch on/off for varying processor indices, for one processor and fifteen hardware traffic generators.



(a) Index 1 (b) Index 6 (c) Index 15

Figure 6: Plots of prefetch on/off for varying processor indices, for the system with sixteen processors.

Figure 5b. This is for the same reason; not all prefetches can be coaleasced to their demand misses.

The lower amount of blocking in the real-world system does allow more prefetch slots to be dispatched, however. Here, similar traffic patterns are still evident; the prefetcher can fully fulfill all prefetches until a point, where prefetches start coaleascing in the tree and causing squashes. This is evident with a delay factor of around 50 for indices 1 and 6, and across the whole range for the processor at index 15. Additionally, the processor at index 15 still exhibits a point at which there is no more available bandwidth for prefetches to be dispatched at a delay factor of around 20.

Another thing of note is that the total speedup is not as great in the combined system as it is in the "full load" system. This is simply due to the reduced system load, and thus memory requests will complete faster with both the prefetcher enabled and disabled. This means that there is less latency for the prefetcher to potentially mask.

6.2 TACLeBench

In addition to the synthetic traffic generators above, the prefetcher was also tested using more real-world benchmarks. These benchmarks were taken from the TACLeBench set of benchmarks [1]. These are a set of benchmarks designed for comparing worst-case execution time evaluation tools, although the lack of any external libraries or external resources also makes them ideal for evaluating systems such as this.

A number of the benchmarks, listed below, were used and evaluated in the same way as the traffic generators, as listed in Section 6.1, that is, they were first evaluated on the "full load" system, then on a system with sixteen processors. In the case of the sixteen-processor system, the same benchmark was replicated multiple times onto each processor. All data (i.e. code, data, heap and BSS) was stored in global memory, with the exception of stack which was placed into local storage.

This set of benchmarks was chosen for the their behaviour when interacting with main memory. They are typically streaming applications, accessing a sufficient amount of memory for real evaluation. In addition, their streaming nature makes them ideal for evaluating a streaming prefetcher, which should be able to capture their streams and speed up their execution.

- *crc*: Cyclic Redundancy Check over 20kB of data.
- *basicmath_small*: Math test routines. Contains large software floating point emulation.
- *rijndael_decoder*: AES decryption of a 32kB file. Contains large straight-line decryption routines.
- *audiobeam*: Beam forming algorithm.
- *gsm_decode*: Decoding of 0.5kB of GSM data.
- *h264dec_ldecode_macroblock*: Decoding a single h264 macroblock.
- *anagram*: Computing anagrams over a 9kB dictionary.
- *sha*: SHA hashing of 32kB of data.

Figure 7 shows the results of these benchmarks when run on the "full load" system (i.e. a single processor with 15 hardware traffic generators), with the single processor in varying processor slots in each system. Each bar denotes the processor at different indices, with the left bar being index 0 and the right being index 15. Here, speedups of 0-50% can be observed, depending upon the behaviour of the benchmark in use.

Many of the smaller benchmarks operating on a stream of data, such as the *crc* and *sha* benchmarks yield a speedup between 0% and 40%. The reasoning for this is twofold; firstly, the code for these benchmarks is reasonably small and so can fit entirely into the instruction cache. This allows the prefetcher to be able to operate solely on the data without

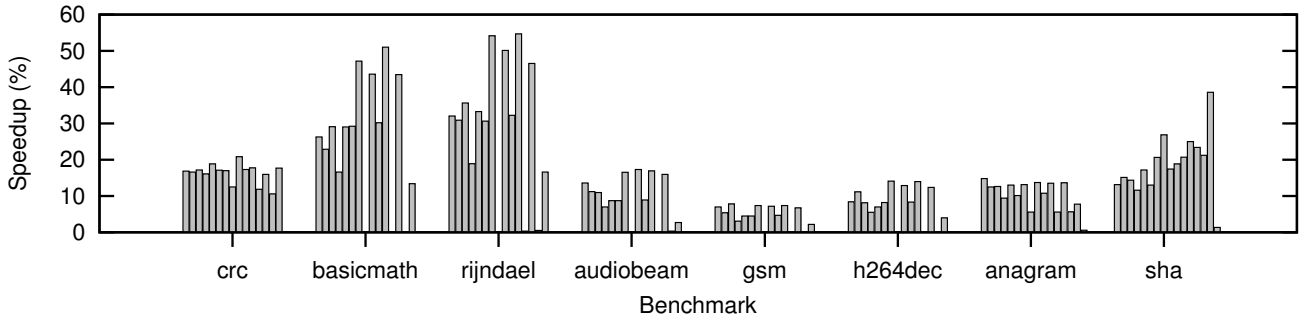


Figure 7: TACLeBench Results for one processor/fifteen hardware traffic generators.

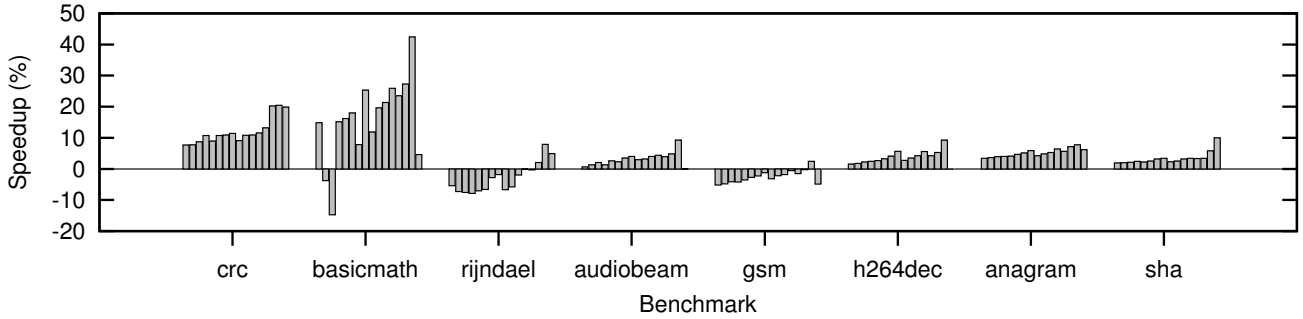


Figure 8: TACLeBench Results for the same task on sixteen processors.

any interference from code prefetches. In addition, there is enough computation for each cache line such that there is sufficient time to dispatch a prefetch.

basicmath_small and *rijndael_decoder* are examples where code prefetch is extremely effective. *basicmath_small* contains a math routine of size 2kB, for example, which the prefetcher can fetch the entirety of accurately. *rijndael_decoder* contains a 8kB block of straight-line code, which can again be accurately prefetched. The speedup for these benchmarks then depends upon the delay after which a prefetch slot can be generated, compared to the delay for which the benchmark would have to dispatch the read manually.

Other benchmarks show different levels of performance. *audiobeam* yields a speedup of 10-20% with the prefetcher enabled, simply because it operates on a stream of data, and that the computation time on this data is sufficiently long that a prefetch can be dispatched within the computation time. *gsm_decode* also operates on a stream of data, although the computation on this data is so long that the prefetcher cannot make a decent impact. There are some large decoding routines which can be prefetched though.

Another interesting note is the performance of the processors at indices 7, 11, 13 and 15. On many of the graphs, these processors show no performance increase with the prefetcher enabled. This is, in part, due to the slots mechanism mentioned earlier. These processors are in a position such that they are on the low-priority side of their first multiplexer. Due to the large amount of blocking that these cores exhibit, this means that it is likely that the processor will be requesting the next required data before a prefetch slot has been dispatched. On other cores, it is more likely that a prefetch slot has been dispatched before the next datum is requested.

It should also be noted that even in these "full load" systems, the presence of the prefetcher does not cause a detriment to system performance; typically it can prefetch useful data, or is not able to fetch any data due to the load on the memory tree.

The aforementioned benchmarks were also run on the sixteen processor system, all executing the same benchmark. The results of this can be found in Figure 8. Again, each bar refers to which processor index is being evaluated, with the left bar being processor index 0 (i.e. the highest priority), and the right being processor index 15 (i.e. the lowest priority).

The benchmarks yield speedups in this system too. All benchmarks tend to exhibit an upwards trend, with the lower priority tasks benefiting more from the prefetcher being enabled. This is because with the prefetcher disabled, these tasks will experience some blocking in line with that in Table 1, but not as much as in the synthetic traffic generator systems. In most cases, the higher-priority tasks will not experience much blocking, and thus there is not as much latency for the prefetcher to hide. The low-priority tasks will still experience a good amount of blocking however, hence there being more latency to potentially be hidden.

The *crc* and *basicmath_small* benchmarks yield a good performance improvement, for example. This is due to similar behaviour to before; the prefetcher can easily prefetch across the whole stream of data, and as mentioned, the lower-priority tasks are favoured since there is more latency to be hidden. *basicmath_small* yields a performance degredation on cores 1 and 2. Given the behaviour of other cores, this is likely because prefetches get dispatched just before the processor dispatches a demand access for those lines, although fails to coaleasce the prefetch and demand access, since there are still some cases for which this cannot happen.

For some tasks, such as *audiobeam* and *h264dec_ldecode*, their computation dominates over the memory delays that they experience. This effect is even more apparent in this system, again, due to the lower delay when communicating with memory. In the *rijndael_decoder* and *gsm_decode*

benchmarks, the execution time is actually *increased*. This effect is due to two things when compared with the traffic generators; first, the time between requests is small (in the case of straight-line code, it will be once every four cycles), and there is more contention over the amount of prefetches available. This leads to a situation where a prefetch may be dispatched too late, which ties up the memory controller with a useless access and thus causes a decrease in the average-case execution time. It is of note, however, that this slower execution time is still faster than those observed in the synthetic "full load" systems discussed previously.

7. CONCLUSIONS & FURTHER WORK

In this paper, we have presented and evaluated a novel method of prefetching within the context of real-time embedded systems, using a system of prefetch feedback "slots". These slots can then take the place of standard memory requests in conventional response-time analysis, allowing a safe method of prefetch that can be utilised within existing response-time analysis frameworks.

This method can yield speedups of up to 50%, depending upon the current traffic patterns of the tasks in the system without causing a detriment to the worst-case execution time of a system, which can subsequently be used to begin to hide the increasing memory delays present in modern multi-core embedded systems.

There are numerous avenues for future work for this system. Our approach has so far only been evaluated using the distributed multiplexer "turns-based" system. This system could, however, be integrated into standard TDM, round-robin, CCSP, or other fair arbitration schemes. Conversely, this approach has only concerned itself with a simple stream prefetcher. Again, it is valid and possible to extend this approach to operate with a different method of prefetching.

8. REFERENCES

[1] TACLeBench, 2013.

[2] A. Agarwal. The Tile Processor : A 64-Core Multicore for Embedded Processing Markets Demanding More Performance. 2007.

[3] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration. *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 3–14, Aug. 2008.

[4] 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*, pages 176–186, New York, New York, USA, 1991. ACM Press.

[5] G. Bernat, A. Colin, and S. Petters. pWCET : a Tool for Probabilistic Worst-Case Execution Time Analysis of Real-Time Systems. pages 1–18, 2003.

[6] H. Cassé and P. Sainrat. OTAWA , a Framework for Experimenting WCET Computations. Number January, pages 1–8, 2006.

[7] K. Chapman. Multiplexer Design Techniques for Datapath Performance with Minimized Routing Resources, 2012.

[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*, pages 62–73, 2002.

[9] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. *ACM SIGPLAN Notices*, 37(10):279, Oct. 2002.

[10] D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee. Response Time Analysis of COTS-Based Multicores Considering the Contention on the Shared Memory Bus. *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, (100202):1068–1075, Nov. 2011.

[11] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2003.

[12] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. *ACM SIGMICRO Newsletter*, 23(1-2):102–110, Dec. 1992.

[13] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):2, 2009.

[14] R. Heckmann and C. Ferdinand. Worst-Case Execution Time Prediction by Static Program Analysis, 2006.

[15] I. Hur and C. Lin. Memory Prefetching Using Adaptive Stream Detection. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 397–408. IEEE, Dec. 2006.

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

[17] 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*, pages 364–373. IEEE Comput. Soc. Press, 1990.

[18] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 261–270. IEEE, Sept. 2009.

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

[20] R. Pellizzoni, A. Schranzhofer, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 741–746, Mar. 2010.

[21] G. Plumbridge, J. Whitham, and N. Audsley. Blueshell : A Platform for Rapid Prototyping of Multiprocessor NoCs and Accelerators. In *Proceedings HEART Workshop*. University of York, 2013.

[22] Rapita. RapiTime Explained, 2014.

[23] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. *Cycle*, pages 128–138, 2000.

[24] 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)*, pages 111–121. IEEE Comput. Soc. Press, 1999.

[25] 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*, pages 63–74, 2007.

[26] S. V. Tota, M. R. Casu, M. R. Roch, L. Rostagno, and M. Zamboni. MEDEA: a hybrid shared-memory/message-passing multiprocessor NoC-based architecture. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 45–50. IEEE, Mar. 2010.