Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 915178, 12 pages
doi:10.1155/2012/915178

Research Article

NCOR: An FPGA-Friendly Nonblocking Data Cache for Soft Processors with Runahead Execution

Kaveh Aasaraai and Andreas Moshovos

Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada

Correspondence should be addressed to Kaveh Aasaraai, aasaraai@eecg.toronto.edu

Received 30 April 2011; Revised 22 August 2011; Accepted 27 August 2011

Academic Editor: Viktor K. Prasanna

Soft processors often use data caches to reduce the gap between processor and main memory speeds. To achieve high efficiency, simple, blocking caches are used. Such caches are not appropriate for processor designs such as Runahead and out-of-order execution that require nonblocking caches to tolerate main memory latencies. Instead, these processors use non-blocking caches to extract memory level parallelism and improve performance. However, conventional non-blocking cache designs are expensive and slow on FPGAs as they use content-addressable memories (CAMs). This work proposes NCOR, an FPGA-friendly non-blocking cache that exploits the key properties of Runahead execution. NCOR does not require CAMs and utilizes smart cache controllers. A 4 KB NCOR operates at 329 MHz on Stratix III FPGAs while it uses only 270 logic elements. A 32 KB NCOR operates at 278 Mhz and uses 269 logic elements.

1. Introduction

Embedded system applications increasingly use soft processors implemented over reconfigurable logic. Embedded applications, like other application classes, evolve over time and their computation needs and structure change. If the experience with other application classes is any indication, embedded applications will evolve and incorporate algorithms with unstructured instruction-level parallelism. Existing soft processor implementations use in-order organizations since these organizations map well onto reconfigurable logic and offer adequate performance for most existing embedded applications. Previous work has shown that for programs with unstructured instruction-level parallelism, a 1-way out-of-order (OoO) processor has the potential to outperform a 2- or even a 4-way superscalar processor in a reconfigurable logic environment [1]. Conventional OoO processor designs are tuned for custom logic implementation and rely heavily on content addressable memories, multiported register files, and wide, multisource and multidestination datapaths. These structures are inefficient when implemented on an FPGA fabric. It is an open question whether it is possible to design an FPGA-friendly soft core that offers the benefits of OoO execution while overcoming the complexity and inefficiency of conventional OoO structures.

A lower complexity alternative to OoO architectures is *Runahead Execution*, or simply *Runahead*, which offers most of the benefits of OoO execution [2]. Runahead relies on the observation that often most of the performance benefits of OoO execution result from allowing multiple outstanding main memory requests. Runahead extends a conventional in-order processor with the ability to continue execution even on a cache miss, with the hope to find more useful misses and thus overlap memory requests. Section 3 discusses Runahead execution in more detail.

Runahead was originally demonstrated on high-end general-purpose systems where main memory latencies are in the order of a few hundred cycles. This work demonstrates that even under the lower main memory latencies (a few tens of cycles) observed in FPGA-based systems today, Runahead can improve performance. However, Runahead, as originally proposed, relies on nonblocking caches which do not map well on FPGAs as they rely on highly associative Content-Addressable Memories (CAMs).

This work presents NCOR (Nonblocking Cache Optimized for Runahead execution) an FPGA-friendly alternative

to conventional nonblocking caches. NCOR does not use CAMs Instead it judiciously sacrifices some of the flexibility of a conventional nonblocking cache to achieve higher operating frequency and thus superior performance when implemented on an FPGA. Specifically, NCOR sacrifices the ability to issue secondary misses, that is, requests for memory blocks that map onto a cache frame with an outstanding request to memory. Ignoring secondary misses enables NCOR to track outstanding misses within the cache frame avoiding the need for associative lookups. This work demonstrates that this simplification does not affect performance nor correctness under Runahead execution.

This work extends the work that introduced NCOR [3] as follows. It quantitatively demonstrates that conventional nonblocking caches are not FPGA-friendly because they require CAMs which lead to a low operating frequency and high area usage. It provides a more detailed description of NCOR and of the underlying design tradeoffs. It explains how NCOR avoids the inefficiencies of conventional designs in more detail. It compares the frequency and area of conventional, nonblocking CAM-based caches and NCOR. Finally, it measures how often secondary misses occur in Runahead execution showing that they are relatively infrequent.

The rest of this paper is organized as follows. Section 2 reviews conventional, CAM-based nonblocking caches. Section 3 reviews Runahead execution and its requirements. Section 4 provides the rationale behind the optimizations incorporated in NCOR. Section 5 presents NCOR architecture. Section 6 discusses the FPGA implementation of NCOR. Section 7 evaluates NCOR comparing it to conventional CAM-based cache implementations. Section 8 reviews related work, and Section 9 summarizes our findings.

2. Conventional Nonblocking Cache

Nonblocking caches are used to extract Memory Level Parallelism (MLP) and reduce latency compared to conventional blocking caches that service cache miss requests one at a time. In blocking caches if a memory request misses in the cache, all subsequent memory requests are blocked and are forced to wait for the outstanding miss to receive data from main memory. Blocked requests may include request for data that is already in the cache or that could be serviced concurrently by modern main memory devices. A nonblocking cache does not block subsequent memory requests when a request misses. Instead, these requests are allowed to proceed concurrently. Some may hit in the cache, while others may be sent to the main memory system. Overall, because multiple requests are serviced concurrently, the total amount of time the program has to wait for the memory to service its requests is reduced.

To keep track of outstanding requests and to make the cache available while a miss is pending, information regarding any outstanding request is stored in Miss Status Holding Registers (MSHRs) [4]. MSHRs maintain the information that is necessary to direct the data received from the main memory to its rightful destination, for example, cache frame or a functional unit. MSHRs can also detect whether a memory request is for a block for which a previous request is still pending. Such requests can be serviced without issuing an additional main memory request. To detect these accesses and to avoid duplicate requests, for every request missing in the cache, the entire array of MSHRs is searched. A matching MSHR means the data has already been requested from memory. Such requests are queued and serviced when the data arrives. Searching the MSHRs requires an associative lookup, which is implemented using a Content-Addressable Memory (CAM). CAMs map poorly to reconfigurable logic as Section 7 shows. As the number of MSHRs bounds the maximum number of outstanding requests, more MSHRs are desirable to extract more MLP. Unfortunately, the area and latency of the underlying CAM grow disproportionately large with the number of MSHRs making large number of MSHRs undesirable.

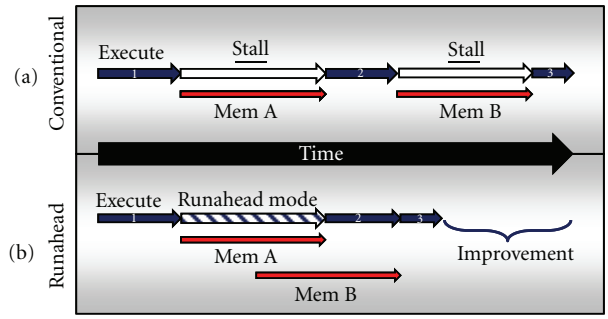

Figure 1: (a) In-order execution of instructions resulting in stalls on cache misses. (b) Overlapping memory requests in Runahead execution.

3. Runahead Execution

Runahead builds on top of a simple in-order processor that maps well onto FPGA fabrics. Runahead improves performance by avoiding the stalls caused by cache misses as Figure 1(a) depicts. A conventional in-order processor stalls whenever a memory request misses in the cache. Even on reconfigurable platforms, a main memory request may take several tens of soft processor cycles to complete limiting performance. Main memory controllers, however, support multiple outstanding requests. Runahead exploits this ability and improves performance by requesting multiple data blocks from memory instead of stalling whenever a request is made. For Runahead to be advantageous the following concerns must be addressed: (1) how can the processor reasonably determine which addresses the program will soon require and (2) how much additional functionality is needed to allow the processor to issue multiple requests.

Runahead shows that it is relatively straightforward to address both issues without overly complicating existing in-order processor designs. As Figure 1(b) shows, upon encountering a cache miss, or a *trigger miss*, instead of stalling the pipeline, the processor continues to execute subsequent independent instructions. This is done with the hope of finding more cache misses to overlap with the trigger miss. Effectively, Runahead uses the program itself to predict near-future accesses that the program will perform and overlaps their retrieval.

When a cache miss is detected, the processor creates a checkpoint of its architectural state (e.g., registers) and enters the *Runahead* execution mode. While the trigger miss is pending, the processor continues executing subsequent independent instructions. Upon the delivery of the trigger miss data, the processor uses the checkpoint and restores all architectural state, so that the results produced in Runahead mode are not visible to the program. The processor then resumes normal execution starting immediately after the instruction that caused the trigger miss.

While all results produced during Runahead mode are discarded, all valid memory requests are serviced by main memory and the data requested is eventually placed in the processor cache. If the program subsequently accesses some of this data, performance may improve as this data was prefetched (i.e., requested earlier from memory). Provided that a sufficient number of instructions independent of the trigger miss are found during runahead mode, the processor has a good chance of reaching other memory requests that miss. As long as a sufficient number of useful memory requests are reached during Runahead mode, performance improves as the processor effectively prefetches these into the cache.

Performance trade-offs with Runahead are complex. On one side, the memory accesses that were initiated during Runahead mode and that fetch useful data effectively prefetch data for subsequent instructions and reduce overall execution time. On the other side, memory accesses that bring useless data pollute the cache and consume memory bandwidth and resources, for example, they may evict useful data from the cache or they may delay another request. Section 7 shows that in practice Runahead execution improves performance.

4. Making a Nonblocking Cache FPGA-Friendly

Runahead execution is conceptually an extension to a simple in-order processor. The simplicity of its architecture is one of the primary reasons that makes Runahead suitable for reconfigurable fabrics. However, for Runahead to be feasible on these fabrics, the extensions must come with low overhead. These extensions include checkpointing and nonblocking caches. Aasaraai and Moshovos proposed a fast, low-cost checkpointing mechanism for out-of-order execution on FPGAs [1]. The same solution could be used in Runahead execution. As Section 7 shows, conventional nonblocking cache designs based on MSHRs do not map well on FPGAs. Accordingly there is a need to design a low-cost nonblocking cache suitable for FPGAs. This work observes that Runahead execution does not need the full functionality of a conventional nonblocking cache and exploits this observation to arrive to an FPGA-friendly nonblocking cache design for Runahead execution.

Conventional nonblocking caches that use MSHRs do not map well on reconfigurable fabrics. The primary reason is that MSHRs use a CAM to perform associative searches. As Section 7 shows MSHRs lead to low clock frequencies and high area usage. In addition to MSHRs, the controller of a nonblocking cache is considerably more complex compared to the one in a blocking cache. The controller is responsible for a wide range of concurrent operations resulting in large, complex state machines. This work presents the Nonblocking Cache Optimized for Runahead execution, or NCOR. NCOR is an FPGA-friendly design that revisits the conventional nonblocking cache design considering the specific needs of Runahead execution. NCOR does away with MSHRs and incorporates optimizations for the cache controller and data storage.

4.1. Eliminating MSHRs. Using the following observations, NCOR eliminates the MSHRs.

(1) As originally proposed, Runahead executes *all* trigger-miss-independent instructions during Runahead mode. However, since the results produced in Runahead mode are later discarded, the processor can choose not to execute some of these instructions as it finds necessary. This option of selective execution can be exploited to reduce complexity by avoiding the execution of instructions that require additional hardware support. One such instruction class is those that cause *secondary misses*, that is, misses on already pending cache frames. Supporting secondary misses is conventionally done via MSHRs, which do not map well to FPGAs.

(2) In most cases servicing secondary misses offers no performance benefit. There are two types of secondary misses: redundant and distinct. A redundant secondary miss requests the same memory block as the trigger miss while distinct secondary miss requests a different memory block that happens to map to the same cache frame as the trigger miss. Section 7 shows that distinct secondary misses are very small fraction of the memory accesses made in Runahead mode.

Servicing a redundant secondary miss cannot directly improve performance further as the trigger miss will bring the data in the cache. A redundant secondary miss may be feeding another load that will miss and that could be subsequently prefetched. However this is impossible as the trigger-miss is serviced first which causes the processor to switch to normal execution. Distinct secondary misses could prefetch useful data, but as Section 7 shows, this has a negligible impact on performance.

Based on these observations the processor can simply discard instructions that cause secondary misses during runahead mode while getting most, and often all, of the performance benefits of runahead execution. However, NCOR still needs to identify secondary misses to be able to discard them. NCOR identifies secondary misses by tracking outstanding misses within the cache frames using a single *pending* bit per frame. Whenever an address misses in the cache, the corresponding cache frame is marked as *pending*. Subsequent accesses to this frame would observe the *pending* bit and will be identified as secondary misses and discarded by the processor. Effectively, NCOR embeds the MHSRs in the cache while judiciously simplifying their functionality to reduce complexity and maintain much of the performance benefits.

4.2. Making the Common Case Fast. Ideally, the cache performs all operations in as few cycles as possible. In particular, it is desirable to service cache hits in a single cycle, as hits are expected to be the common case. In general, it is desirable to design the controller to favor the frequent operations over the infrequent ones. Accordingly, NCOR uses a three-part cache controller which favors the most frequent requests, that is, cache hits, by dedicating a simple subcontroller just for hits. Cache misses and all noncacheable requests (e.g., I/O requests) are handled by other sub-controllers which are triggered exclusively for such events and are off the critical path for hits. These requests complete in multiple cycles. The next section explains the NCOR cache controller architecture in detail.

5. NCOR Architecture

Figure 2 depicts the basic structure of NCOR. The cache controller comprises *Lookup*, *Request*, and *Bus* components. NCOR also contains *Data*, *Tag*, *Request,* and *Metadata* storage units.

5.1. Cache Operation. NCOR functions as follows.

(i) *Cache Hit*. The address is provided to *Lookup* which determines, as explained in Section 5.2, that this request is a hit. The data is returned in the same cycle for load operations and is stored in the cache during the next cycle for store operations. Other soft processor caches, such as those of Altera Nios II, use two cycles for stores as well [5].

(ii) *Tirgger Cache Miss*. If *Lookup* identifies a cache miss, it sends a signal to *Request* to generate the necessary requests to handle the miss. *Lookup* blocks the cache interface until *Request* signals back that it has generated all the necessary requests.

Request generates all the necessary requests directed at *Bus* to fulfill the pending memory operation. If a dirty line must be evicted, a write-back request is generated first. Then a cache line read request is generated and placed in the *Queue* between *Request* and *Bus*.

Bus receives requests through the *Queue* and sends the appropriate signals to the system bus. The *pending bit* of the cache frame that will receive the data is set.

(iii) *Secondary Cache Miss in Runahead Mode*. If *Lookup* identifies a secondary cache miss, that is, a miss on a cache frame with *pending* bit set, it discards the operation.

(iv) *Secondary Cache Miss in Normal Mode*. If *Lookup* identifies a secondary cache miss in normal execution mode, it blocks the pipeline until the frame's *pending* bit is cleared. It is possible to have a secondary miss in normal execution mode as a memory access initiated in Runahead mode may be still pending. In normal execution processor cannot discard operations and must wait for the memory request to be fulfilled.

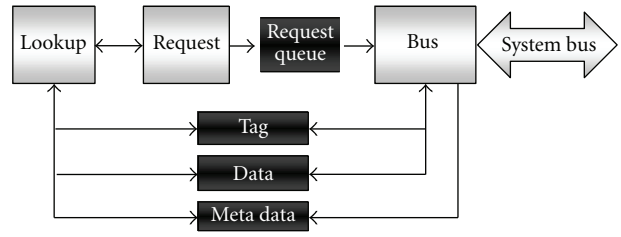

Figure 2: Nonblocking cache structure.

The following subsections describe the function of each NCOR component.

5.2. Lookup. *Lookup* is the cache interface that communicates with the processor and receives memory requests. *Lookup* performs the following operations.

(i) For cache accesses, *Lookup* compares the request address with the tag stored in the *Tag* storage to determine whether this is a hit or a miss.

(ii) For cache hits, on a load, *Lookup* reads the data from the *Data* storage and provides it to the processor in the same cycle as the *Tag* access. Reading the *Data* storage proceeds in parallel with the *Tag* access and comparison. Stores, on the other hand, take two cycles to complete as writes to the *Data* storage happen in the cycle after the hit is determined. In addition, the cache line is marked as dirty.

(iii) For cache misses, *Lookup* marks the cache line as pending.

(iv) For cache misses and non-cacheable requests, *Lookup* triggers *Request* to generate the appropriate requests. In addition, for loads, it stores the instruction metadata in the *MetaData* storage. *Lookup* blocks the processor interface until *Request* signals it has generated all the necessary requests.

(v) For cache accesses, whether the request hits or misses in the cache, if the corresponding cache line is *pending*, *Lookup* discards the request if the processor is in Runahead mode. However, if the processor is in normal execution mode, *Lookup* stalls the processor.

5.3. Request. *Request* is normally idle waiting for a trigger from *Lookup*. When triggered, it issues the appropriate requests to *Bus* through request *Queue*. *Request* performs the following operations.

(i) Waits in the idle state until triggered by *Lookup*.

(ii) For cache misses, *Request* generates a cache line read request. In addition if the evicted line is dirty, *Request* generates a cache line write-back request.

(iii) For non-cacheable requests, depending on the operation, *Request* generates a single read or write request.

(iv) When all necessary requests are generated and queued, *Request* notifies *Lookup* of its completion and returns to its idle state.

5.4. Bus. *Bus* is responsible for servicing bus requests generated by *Request*. *Bus* receives requests through the request *Queue* and communicates through the system bus with the main memory and peripherals. *Bus* consists of two internal modules.

5.4.1. Sender. *Sender* sends requests to the system bus. It removes requests from the request *Queue* and, depending on the request type, sends the appropriate signals to the system bus. A request can be of one of the following types.

(i) *Cache Line Read*. Read requests are sent to the system bus for each data word of the cache line. The critical word (word originally requested by the processor) is requested first. This ensures minimum wait time for data delivery to the processor.

(ii) *Cache Line Write-Back*. Write requests are sent to the system bus for each data word of the dirty cache line. Data words are retrieved from *Data* storage and sent to the system bus.

(iii) *Single Read/Write*. A single read/write request is sent to the memory/peripheral through the system bus.

5.4.2. Receiver. *Receiver* handles the system bus responses. Depending on the processor's original request type, one of the following actions is taken.

(i) *Load from Cache.* Upon receipt of the first data word, *Receiver* signals request completion to the processor and provides the data. This is done by providing the corresponding metadata from the *MetaData* storage to the processor. *Receiver* also stores all the data words received in the *Data* storage. Upon receipt of the last word, it stores the cache line tag in the corresponding entry in the *Tag* storage, sets the valid bit, and clears both dirty and *pending* bits.

(ii) *Store to Cache*. The first data word received is the data required to perform the store. Receiver combines the data provided by the processor with the data received from the system bus and stores it in the *Data* storage. It also stores subsequent data words, as they are received, in the *Data* storage. Upon the receipt of the last word, *Receiver* stores the cache line tag in the corresponding entry in the *Tag* storage, sets both valid and dirty bits, and clears the *pending* bit.

(iii) *Noncacheable Load*. Upon receipt of the data word, *Receiver* signals request completion to the processor and provides the data. It also provides the corresponding metadata from the *MetaData* storage.

5.5. Data and Tag Storage. The *Data* and *Tag* storage units are tables holding cache line data words, tags, and status bits. *Lookup* and *Bus* both access *Data* and *Tag*.

5.6. Request Queue. Request *Queue* is a FIFO memory holding requests generated by *Request* directed at *Bus*. Request *Queue* conveys requests in the order they are generated.

5.7. MetaData. For outstanding load requests, that is, load requests missing in the cache or non-cacheable operations, the cache stores the metadata accompanying the request. This data includes Program Counter and destination register for load instructions. Eventually when the request is fulfilled this information is provided to the processor along with the data loaded from the memory or I/O. This information allows the loaded data to be written to the register file. *MetaData* is designed as a queue so that requests are processed in the order they were received. No information is placed in the *MetaData* for Stores as the processor does not require acknowledgments for their completion.

6. FPGA Implementation

This section presents the implementation of the nonblocking cache on FPGAs. It discusses the design challenges and the optimizations applied to improve clock frequency and minimize the area. It first discusses the storage organization and usage and the corresponding optimizations. It then discusses the complexity of the cache controller's state machine and how its critical path was shortened for the most common operations.

6.1. Storage Organization. Modern FPGAs contain dedicated Block RAM (BRAM) storage units that are fast and take significantly less area compared to LUT-based storage. This subsection explains the design choices that made it possible to use BRAMs for most of the cache storage components.

6.1.1. Data. Figure 3 depicts the *Data* storage organization. As BRAMs have a limited port width, the entire cache line does not fit in one entry. Consequently, cache line words are spread, one word per entry, over multiple BRAM entries. This work targets the Nios-II ISA [5] which supports byte, half-word, and word stores (one, two, and four, bytes resp.). These are implemented using the BRAM *byte enable* signal [6]. Using this signal avoids two-stage writes (read-modify-write) which would increase area due to the added multiplexers.

6.1.2. Tag. Figure 3 depicts the *Tag* storage organization. Unlike cache line data, a tag fits in one BRAM entry. In order to reduce BRAM usage, we store cache line status bits, that is, valid, dirty, and *pending* bits, along with the tags.

Despite the savings in BRAM usage by storing cache line status bits along with the tags, the following problem arises. *Lookup* makes changes only to the *dirty* and *pending* bits and should not alter *valid* or *Tag* bits. In order to preserve *valid* and *Tag* bits while performing a write, a two-stage write could be used, in which bits are first read and then written back. This read-modify-write sequence increases area and complexity and hurts performance. We overcome this problem by using the *byte enable* signals. As Figure 3 shows, we

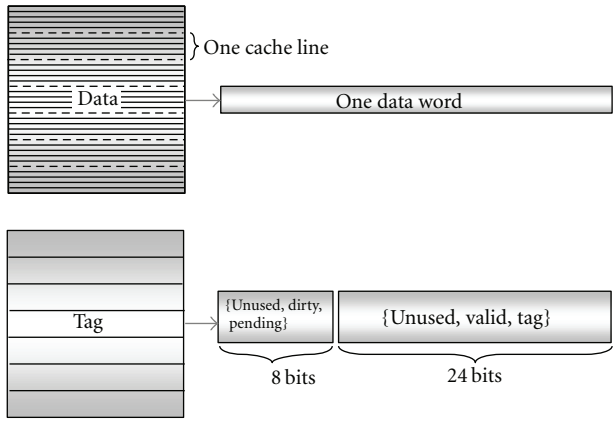

Figure 3: The organization of the *Data* and *Tag* storage units.

store *valid* and *Tag* bits in the lower 24 bits and *dirty* and *pending* bits in the higher eight bits. Then using the *byte enable* signal, *Lookup* is able to change only the *dirty* and *pending* bits.

6.2. BRAM Port Limitations. Although BRAMs provide fast and area-efficient storage, they have a limited number of ports. A typical BRAM in today's FPGAs has one read and one write port [6]. Figure 4 shows that both *Lookup* and *Bus* write and read to/from the *Data* and *Tag* storages. This requires four ports. Our design uses only two ports based on the following observations: BRAMs can be configured to provide two ports, each providing both write and read operations over one address line. Although *Lookup* and *Bus* both write and read to/from the *Data* and *Tag* at the same time, each only requires one address line.

6.2.1. Tag. For every access from *Lookup* to the *Tag* storage, *Lookup* reads the *Tag*, *valid*, *dirty,* and *pending* bits for a given cache line. *Lookup* also writes to the *Tag* storage in order to mark a line dirty or *pending*. However, reads and writes never happen at the same time as marking a line dirty (for stores) or pending (for misses) happens one cycle after the tag and other status bits are read. *Bus* only writes to the *Tag* storage, when a cache line is retrieved from the main memory. Therefore, dedicating one address line to *Lookup* and one to *Bus* is sufficient to access the *Tag* storage.

6.2.2. Data. For every *Lookup* access to the *Data* storage, *Lookup* either reads or writes a single, or part of a, word. However, *Bus* may need to write to or read from the *Data* storage at the same time. This occurs if *Bus* is sending words of a write-back request while a previously requested data is being delivered by the system bus. To avoid this conflict, we restrict *Bus* to send a write-back data word only when the system bus is not delivering any data. Forward progress is guaranteed as outstanding write-back requests do not block responses from the system bus. This restriction minimally impacts cache performance as words are sent as soon as the system bus is idle. In Section 7.9 we show that even in the worst case scenario, impact on performance is marginal. With this modification, dedicating one address line to *Lookup* and one to *Bus* is sufficient for accessing the *Data* storage.

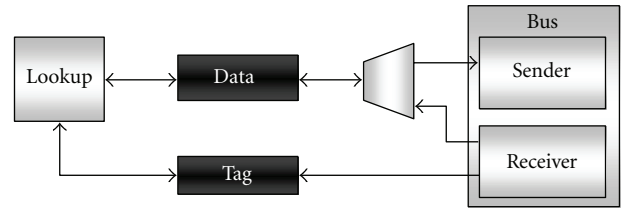

Figure 4: Connections between Data and Tag storages and Lookup and Bus components.

6.3. State Machine Complexity. The cache controller is responsible for looking up in the cache, performing loads and stores, generating, bus requests and handling bus transactions. Given the number of operations that the controller handles, in many cases concurrently, it requires a large and complex state machine. A centralized cache controller can be slow and has the disadvantage of treating all requests the same. However, we would like the controller to respond as quickly as possible to those requests that are most frequent, that is, requests that hit in the cache. Accordingly, we partition the controller into subcomponents. One could partition the controller into two components of CPU-side and bus-side, as Figure 5(a) shows. The CPU-side component would be responsible for looking up addresses in the cache, performing loads and stores, handling misses and non-cacheable operations, and sending necessary requests to the bus-side component. The bus-side component would communicate with the main memory and system peripherals through the system bus.

Due to the variety of operations that the CPU-side component is responsible for, we find that it still requires a non-trivial state machine. The state machine has numerous input signals, and this reduces performance. Among its inputs is the cache hit/miss signal, a time-critical signal due to the large comparator used for tag comparison. As a result, implementing the CPU-side component as one state machine leads to a long critical path.

Higher operating frequency is possible by further partitioning the CPU-side component into two subcomponents, *Lookup* and *Request*, which cooperatively perform the same set of operations. Figure 5(b) depicts the three-component cache controller. The main advantage of this controller is that cache lookups that hit in the cache, the most frequent operations, are handled only by *Lookup* and are serviced as fast as possible. However, this organization has its own disadvantages. In order for the *Lookup* and *Request* to communicate, for example, in the case of cache misses, extra clock cycles are required. Fortunately, these actions are relatively rare. In addition, in such cases servicing the request takes in the order of tens of cycles. Therefore, adding one extra cycle delay to the operation has little impact on performance. Figure 6 shows an overview of the two state machines corresponding to Lookup and Request.

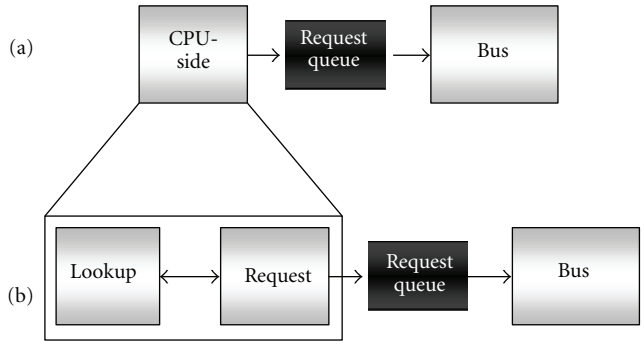

Figure 5: (a) Two-component cache controller. (b) Three-component cache controller.

6.4. Latching the Address. We use BRAMs to store data and tags in the cache. As BRAMs are synchronous rams, the input address needs to be available just before the appropriate clock edge (rising in our design) of the cycle when cache lookup occurs. Therefore, in a pipelined processor, the address has to be forwarded to the cache from the previous pipeline stage, for example, the execute stage in a typical 5-stage pipeline. After the first clock edge, the input address to the cache changes as it is forwarded from the previous pipeline stage. However, the input address is further required for various operations, for example, tag comparison. Therefore, the address must be latched.

Since some cache operations take multiple cycles to complete, the address must be latched only when a new request is received. This occurs when *Lookup* 's state machine is *entering* the *lookup state*. Therefore, the address register is clocked based on the *next state* signal. This is a time-critical signal, and using it to clock a wide register, as is the case with the address register, negatively impacts performance.

To avoid using this time-critical signal we make the following observations. The cache uses a latched address in two phases: in the first cycle for tag comparison and in subsequent cycles for writes to *Data* storage and request generations. Accordingly, we can use two separate registers, *addr_always* and *addr_lookup* one per phase. At every clock cycle, we latch the input address into *addr_always*. We use this register for tag comparison in the first cycle. At the end of the first cycle, if *Lookup* is *in* the lookup state, we copy the content of *addr_always* into *addr_lookup*. We use this register for writes to the cache and request generation. As a result, the *addr_always* register is unconditionally clocked every cycle. Also, we use *Lookup*'s *current state* register, rather than its *next state* combinational signal, to clock the *addr_lookup* register. This improves the design's operating frequency.

7. Evaluation

This section evaluates NCOR. It first compares the area and frequency of NCOR with those of a conventional MSHR-based nonblocking cache. It then shows the potential performance advantage that Runahead execution has over an in-order processor using a nonblocking cache.

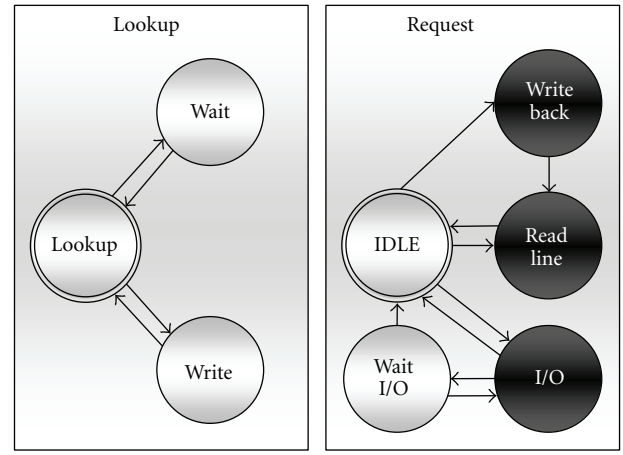

Figure 6: Lookup and Request state machines. Double-lined states are initial states. Lookup waits for Request completion in the "wait" state. All black states generate requests targeted at the Bus controller.

7.1. Methodology. The processor performance is estimated using a cycle-accurate, Nios II full system simulator capable of booting and running the uCLinux operating system [7]. The simulated processor models include a 5-stage in-order pipelined processor, 1- to 4-way superscalar processors, and Runahead processors. Table 1 details the simulated processor microarchitectures.

This evaluation uses benchmarks from the SPEC CPU 2006 suite which is typically used to evaluate desktop system performance [8]. These benchmarks are representative of applications that have unstructured ILP. The assumption is that in the future, soft-core-based systems will be called upon to run demanding applications that will exhibit sufficiently similar behavior. Those benchmarks that the uCLinux's toolchain cannot compile or that require a floating-point unit are excluded. Each benchmark run uses a reference input. These inputs are stored in main memory and are accessed through the block device driver (Ram-disk). Measurements are taken for a sample of one billion instructions after skipping several billions of instructions so that execution is past initialization.

Area and frequency are measured over Verilog implementations. Quartus II v10.0 did synthesis and place-and-routing and the target FPGA was the Altera Stratix III EP3SL150F1152C2. NCOR is compared against a conventional, MSHR-based nonblocking cache.

7.2. Simplified MSHR-Based Nonblocking Cache. NCOR was motivated as a higher-speed, lower-cost and complexity alternative to conventional, MSHR-based nonblocking caches. A comparison of the two designs is needed to demonstrate the magnitude of these advantages. Our experience has been that the complexity of a conventional nonblocking cache design quickly results in an impractically slow and large FPGA implementation. This makes it necessary to seek FPGA-friendly alternatives such as NCOR. For the purposes of demonstrating that NCOR is faster and smaller than a conventional nonblocking cache, it is sufficient to compare against a simplified nonblocking cache. This is sufficient, as long as

Table 1: Architectural properties of simulated processors.

I-cache size (Bytes)	32 K
D-Cache size (Bytes)	4–32 K
Cache line size	32 bytes
Cache associativity	Direct mapped
Memory latency	26 cycles
BPredictor type	GShare
BPredictor entries	4096
BTB entries	256
Pipeline stages	5
No. of outstanding misses	32

the results demonstrate the superiority of NCOR and provided that the simplified conventional cache is clearly faster and smaller than a full-blown conventional nonblocking cache implementation. The simplifications made to the conventional MSHR nonblocking cache are as follows.

(i) Requests mapping to a cache frame for which a request is already pending are not supported. Allowing multiple pending requests targeting the same cache frame substantially increases complexity.

(ii) Each MSHR entry tracks a single processor memory request as opposed to all processor requests for the same cache block request [4]. This eliminates the need for a request queue per MSHR entry which tracks individual processor requests, some of which may map onto the same cache block. In this organization the MSHRs serve as queues for both pending cache blocks and processor requests. Secondary misses are disallowed.

(iii) Partial (byte or half-word) loads/stores are not supported.

We use this simplified MSHR-based cache for FPGA resource and clock frequency comparison with NCOR. In the performance simulations we use a regular MSHR-based cache.

7.3. Resources. FPGA resources include ALUTs, block RAMs (BRAM), and the interconnect. In these designs interconnect usage is mostly tied to ALUT and BRAM usage. Accordingly, this section compares the ALUT and BRAM usage of the two cache designs. Figure 7 reports the number of ALUTs used by NCOR and the MSHR-based cache for various capacities. The conventional cache uses almost three times as many ALUTs compared to NCOR. There are two main reasons why this difference occurs. (1) The MSHRs in the MSHR-based cache use ALUTs exclusively instead of a mix of ALUTs and blockrams. (2) The large number of comparators required in the CAM structure of the MSHRs require a large number of ALUTs.

While the actual savings in the number of ALUTs are small compared to the number of ALUTs found in high capacity FPGAs available today (>100 K ALUTs) such small

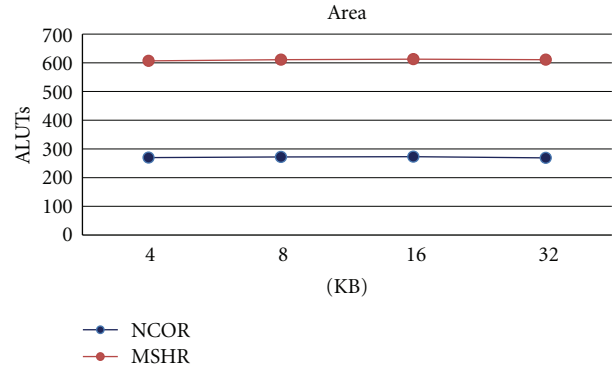

Figure 7: Area comparison of NCOR and MSHR-based caches over various capacities.

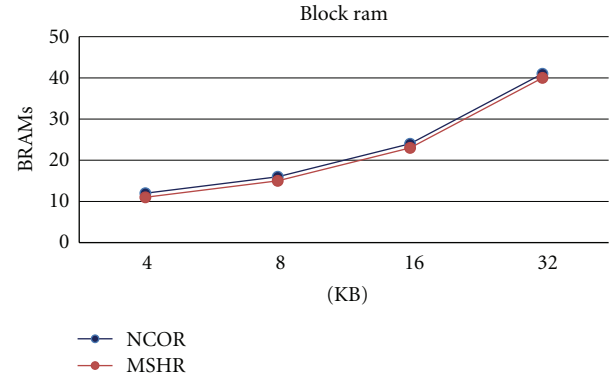

Figure 8: BRAM usage of NCOR and MSHR-based caches over various capacities.

savings add up, for example, in a multiprocessor environment. Additionally, there are designs where low-capacity FPGAs such in, for example, low budget, low power, or military applications.

In NCOR, the bulk of the cache is implemented using BRAMs, hence the low area usage of the cache. The vast majority of the BRAMs contain the cache's data, tag, and status bits. As expected, both caches experience a negligible change in ALUT usage over different capacities, as most of the cache storage is implemented using BRAMs.

Figure 8 shows the number of BRAMs used in each cache for various capacities. Compared to the conventional cache, NCOR uses one more BRAM as it stores pending memory requests in BRAMs rather than in MSHRs.

7.4. Frequency. Figure 9 reports the maximum clock frequency the NCOR and the MSHR-based cache can operate at and for various capacities. NCOR is consistently faster. The difference is at its highest (58%) for the 4 KB caches with NCOR operating at 329 MHz compared to the 207 MHz for the MHSR-based cache. For both caches, and in most cases, frequency decreases as the cache capacity increases. At 32 KB NCOR's operating frequency is within 18% of the 4 KB NCOR. While increased capacity results in reduced frequency in most cases, the 8 KB MSHR-based cache is faster

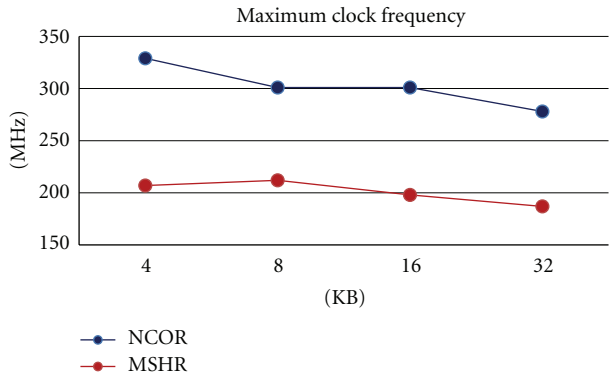

Figure 9: Clock frequency comparison of NCOR and of a four-entry MSHR-based cache over various cache capacities.

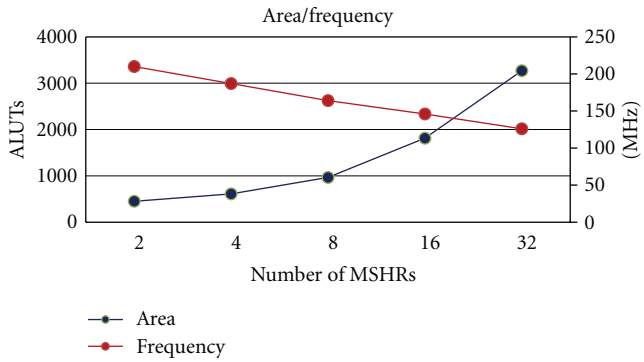

Figure 10: Area and clock frequency of a 32 KB MSHR-based cache with various number of MSHRs. The left axis is ALUTs and the right axis is clock frequency.

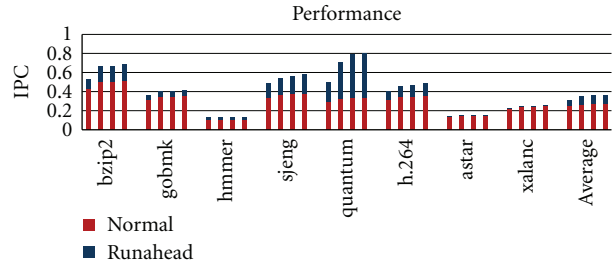

Figure 11: Speedup gained by Runahead execution on 1- to 4-way superscalar processors. The lower parts of the bars show the IPC of the normal processors. The full bars show the IPC of the Runahead processor.

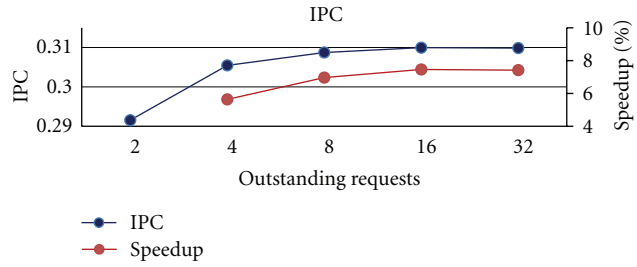

Figure 12: The impact of number of outstanding requests on IPC. Speedup is measured over the first configuration with two outstanding requests.

than its 4 KB counterpart. As the cache capacity increases, more sets are used, and hence the tag size decreases. Accordingly, this makes tag comparisons faster. At the same time, the rest of the cache becomes slower. These two latencies combine to determine the operating frequency which is at a local minimum at a capacity of 8 KB for the MSHR-based cache. However, as cache capacity continues to grow, any reduction in tag comparison latency is overshadowed by the increase in latency in other components.

7.5. MSHR-Based Cache Scalability. The NCOR studied in this work is capable of handling up to 32 outstanding requests. Supporting more outstanding requests in NCOR basically comes for free as these are tracked in BRAMs. An MSHR-based cache however uses CAMs, hence LUTs for storage. Figure 10 reports how the frequency and area of the MSHR-based cache scale with MSHR entry count. As expected, as the number of MSHRs increases clock frequency drops and area increases. With 32 MSHRs, the MHSR-based cache operates at only 126 MHz and requires 3269 ALUTs.

7.6. Runahead Execution. Figure 11 reports the speedup achieved by Runahead execution on 1- to 4-way superscalar processors. For this comparison, performance is measured as the instructions per cycle (IPC) rate. IPC is a frequency-independent metric and thus is useful in determining the range of frequencies for which an implementation can operate on and still outperform an alternative. Runahead is able to outperform the corresponding in-order processor by extracting memory-level parallelism effectively hiding the high main memory latency. For a typical single-issue pipeline (1-way), on average, Runahead improves IPC by 26%.

As the number of outstanding memory requests increases, higher memory level parallelism is extracted, hence higher performance. Figure 12 shows how the IPC scales with increasing the number of outstanding requests. Moving from two outstanding requests to 32, we gain, on average, 7% in IPC. The impact of the number of outstanding requests is even greater as the memory latency increases. We study memory latency impact in Figure 13. When memory latency is lower, increasing outstanding requests marginally increases speedup, that is, by 7%. However with a high memory latency, by moving from two outstanding requests to 32, the speedup doubles, that is, from 26% to 54%.

Next we compare the speedup gained with NCOR to that of a full-blown MSHR-based cache. Figure 14 compares the IPC of Runahead execution with NCOR and MSHR-based caches. NCOR achieves slightly lower IPC, less than 4% on average, as it sacrifices memory level parallelism for lower complexity. However, in the case of sjeng, MSHR performs worse. MSHR is more aggressive in prefetching cache lines and in this case pollutes the cache rather than prefetching useful data.

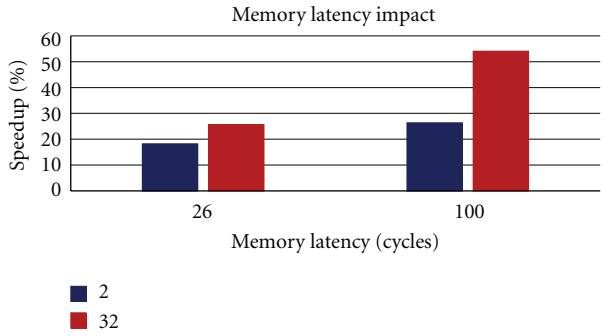

Figure 13: Speedup gained by Runahead execution with two and 32 outstanding requests, with memory latency of 26 and 100 cycles.

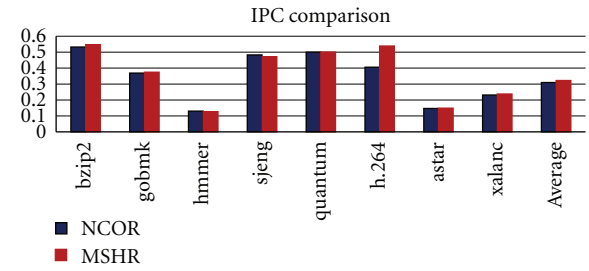

Figure 14: Performance comparison of Runahead with NCOR and MSHR-based cache.

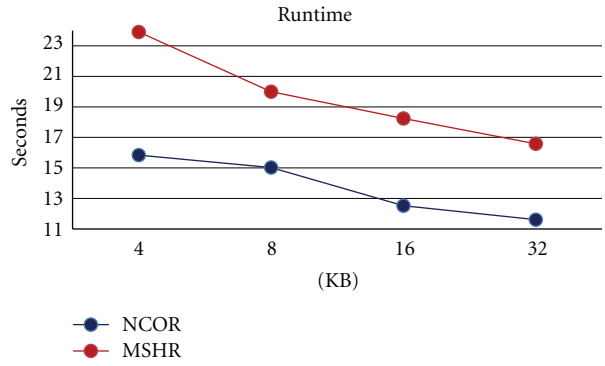

Figure 15: Average runtime in seconds for NCOR and MSHR-based cache.

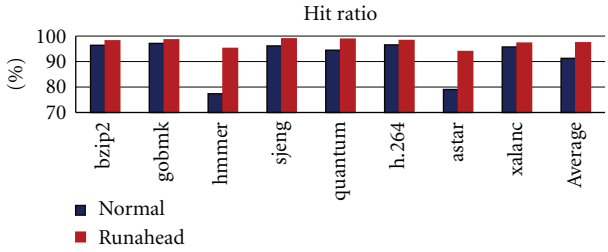

Figure 16: Cache hit ratio for both normal and Runahead execution.

Finally we compare NCOR and MSHR-based caches based on both IPC and their operating frequency. Figure 15 compares the two caches in terms of runtime in seconds over a range of cache sizes. NCOR performs the same task up to 34% faster than MSHR. It should be noted that NCOR with 4 KB capacity performs faster than a 32 KB MSHR-based cache.

7.7. Cache Performance. This section compares cache performance with and without Runahead execution. Figure 16 reports hit ratio for a cache with 32 KB capacity with and without Runahead execution. Runahead improves cache hit ratio, by as much as 23% for hmmer and by about 7% on average. We also report the number of cache Misses Per Kilo Instructions (MPKIs) in Figure 17. Runahead reduces MPKI, on average by 39% as it effectively prefetches useful data into the cache.

7.8. Secondary Misses. Runahead execution tied with NCOR achieves high performance even though the cache is unable to service secondary misses. This section provides additional insight on why discarding secondary misses has little effect on performance. Figure 18 reports, on average, how many times the cache observes a secondary miss (only misses to a different memory block) while in Runahead mode. The graph shows that on every invocation of Runahead mode only 0.1 secondary misses are seen, on average over all benchmarks. Even if the cache was able to service secondary misses, it would have generated only 10 memory requests every 100 times that it switches to Runahead mode. Therefore, discarding secondary misses does not take away a significant opportunity to overlap memory requests. Even for hmmer which experiences a high number of secondary misses, Runahead achieves a 28% speedup as Figure 11 reports. This shows that nonsecondary misses are in fact fetching useful data.

7.9. Write-Back Stall Effect. In Section 6.2.2 we showed that BRAM port limitation requires NCOR to delay write-backs in case the system bus is responding to an earlier cache line read request. Unfortunately in order to study the impact on IPC we need a very accurate DDR-2 model in software simulations which our infrastructure does not include. Alternatively, we study the most pessimistic scenario in which all write-backs coincide with data return of pending cache line reads, resulting in write-back stalls. Figure 19 shows that, even in this worst case scenario, still Runahead execution with NCOR is effective, and on average less than 2% performance is lost.

8. Related Work

Related work in soft processor cache design includes work on automatic generation of caches and synthesizable high performance caches, including nonblocking and traversal caches. To the best of our knowledge, NCOR is the first FPGA-friendly nonblocking data cache optimized for Runahead execution.

Runahead execution requires a checkpoint mechanism for the processor's architectural state. Aasaraai and Moshovos

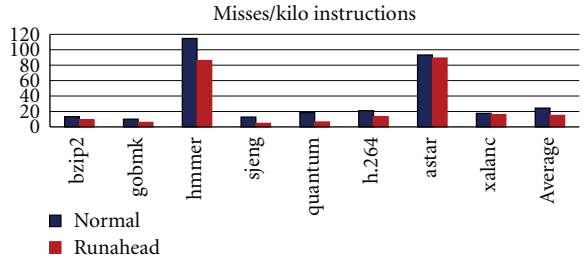

Figure 17: Number of misses per 1000 instructions executed in both normal and Runahead execution.

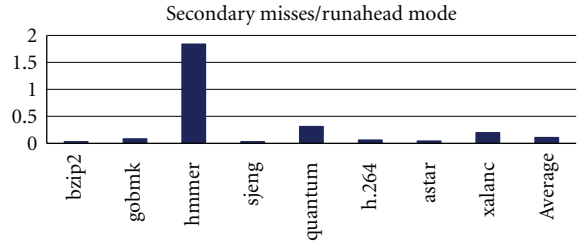

Figure 18: Average number of secondary misses (misses only to different cache blocks) observed per invocation of Runahead executions in a 1-way processor.

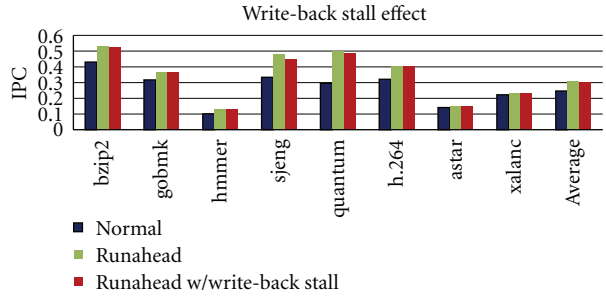

Figure 19: IPC comparison of normal, Runahead and Runahead with worst case scenario for write-back stalls.

proposed CFC, a copy-free checkpointing mechanism which can be used to support Runahead execution [1]. CFC avoids data copying by adding a level of indirection to the checkpointing mechanism. CFC achieves superior performance compared to the conventional method of copying data. CFC, and checkpointing in general, is used in various architectures, for example, out-of-order execution, transactional memory, and Runahead execution.

Yiannacouras and Rose created an automatic cache generation tool for FPGAs [9]. Their tool is capable of generating a wide range of caches based on a set of configuration parameters, for example, cache size, associativity, latency, and data width. The tool is also useful in identifying the best cache configuration for a specific application.

PowerPC 470S is a synthesizable soft-core implementation that is equipped with nonblocking caches. This core is available under a nondisclosure agreement from IBM [10]. A custom logic implementation of this core, PowerPC 476FP, has been implemented by LSI and IBM [10].

Stitt and Coole present a traversal data cache framework for soft processors [11]. Traversal caches are suitable for applications with pointer-based data structures. It is shown that, using traversal caches, for such applications performance may improve by as much as 27x. Traversal caches are orthogonal to NCOR.

9. Conclusion

This work presented NCOR, an FPGA-Friendly nonblocking data cache implementation for soft processors with Runahead execution. It showed that a conventional nonblocking cache is expensive to build on FPGAs due to the CAM-based structures used in its design. NCOR exploits the key properties of Runahead execution to avoid CAMs and instead stores information about pending requests inside the cache itself. In addition, the cache controller is optimized by breaking its large and complex state machine into multiple, smaller, and simpler subcontrollers. Such optimizations improve design operating frequency. A 4 KB NCOR operates at 329 MHz on Stratix III FPGAs while it uses only 270 logic elements. A 32 KB NCOR operates at 278 Mhz using 269 logic elements.

Acknowledgments

This work was supported by an NSERC Discovery grant and equipment donations from Altera Corp. K. Aasaraai was supported by an NSERC-CGS scholarship.

References

[1] K. Aasaraai and A. Moshovos, "Towards a viable out-of-order soft core: copy-free, checkpointed register renaming," in *the 19th International Conference on Field Programmable Logic and Applications (FPL '09)*, Prague, Czech Republic, September 2009.

[2] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *Proceedings of the International Conference on Supercomputing*, pp. 68–75, July 1997.

[3] K. Aasaraai and A. Moshovos, "An efficient non-blocking data cache for soft processors," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs*, December 2010.

[4] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pp. 81–87, 1982.

[5] Altera Corp., "Nios II Processor Reference Handbook v10.0," 2010.

[6] Altera Corp, "Stratix III Device Handbook: Chapter 4. TriMatrix Embedded Memory Blocks in Stratix III Devices," 2010.

[7] Arcturus Networks Inc, "uClinux," http://www.uclinux.org/.

[8] Standard Performance Evaluation Corporation, "SPEC CPU 2006," http://www.spec.org/cpu2006/.

[9] P. Yiannacouras and J. Rose, "A parameterized automatic cache generator for FPGAs," in *Proceedings of Field-Programmable Technology (FPT)*, pp. 324–327, 2003.

[10] IBM and LSI, "PowerPC 476FP Embedded Processor Core and PowerPC 470S Synthesizable Core User's Manual," http://www-03.ibm.com/press/us/en/pressrelease/28399.wss.

[11] G. Stitt and J. Coole, "Traversal caches: a framework for FPGA acceleration of pointer data structures," *International Journal of Reconfigurable Computing*, vol. 2010, Article ID 652620, 16 pages, 2010.