An Efficient Non-Blocking Data Cache for Soft Processors

Kaveh Aasaraai and Andreas Moshovos
Department of Electrical and Computer Engineering
University of Toronto
Toronto, ON, Canada
{*aasaraai, moshovos*}*@eecg.toronto.edu*

***Abstract*—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 non-blocking caches to tolerate main memory latencies. Conventional non-blocking caches are expensive and slow on FPGAs as they use content-addressable memories (CAMs). This work exploits key properties of runahead execution and demonstrates an FPGA-friendly non-blocking cache design that does not require CAMs. A non-blocking 4KB cache operates at 329MHz on Stratix III FPGAs while it uses only 270 logic elements. A 32KB non-blocking cache operates at 278Mhz and uses 269 logic elements.**

***Keywords*-Soft Processor; Data Cache; Non-Blocking; Runahead**

I. Introduction

Soft processors implemented over reconfigurable logic are increasingly being used in embedded system applications. Historically, applications evolve in their computation needs and structure. Embedded applications are not immune to this trend. Accordingly, it is likely that soft processors will be called upon to execute applications with unstructured instruction level parallelism. Previous work has shown that for such programs, a 1-way OoO processor in an FPGA environment has the potential to outperform a 2- or even a 4-way superscalar processor [1]. Unfortunately, conventional OoO processor implementations are tuned for custom logic implementation and rely heavily on content addressable memories, multiported register files, and wide, multi-source and multi-destination datapaths. Such structures exhibit poor efficiency when implemented in 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 without the existing complexities and inefficiencies.

Previous work has shown that Runahead execution offers most of the benefits of OoO execution while avoiding much of its complexity on custom implementations [2]. Runahead relies on the observation that 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 when a memory operation misses in the cache, with the hope to find more useful misses and thus overlap memory requests.

Originally runahead was demonstrated for 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 relatively low main memory latencies (a few tens of cycles) observed in FPGA-based systems today, runahead execution still offers most of the OoO execution performance benefits. Having demonstrated the potential of runahead execution, this work proceeds to present a non-blocking data cache design, a key component of a runahead architecture.

Conventional non-blocking caches are not FPGA-friendly as they rely on highly-associative content-addressable memories (CAMs). This work proposes a non-blocking cache design that does not use CAMs. The design judiciously sacrifices some of the flexibility of a conventional non-blocking cache in order to achieve higher operating frequency and superior performance when implemented on an FPGA. Specifically, the proposed cache sacrifices the ability to issue secondary misses, that is requests for memory blocks that map onto a cache line with an outstanding request to memory (this includes requests for the same block). Doing so enables the cache to track outstanding misses within the cache line avoiding the need for associative lookups. We demonstrate that this simplification does not affect performance nor correctness under runahead execution.

The rest of this paper is organized as follows: Section II provides background on non-blocking caches and Runahead execution. Section III presents the architecture of our non-blocking cache design. Section IV discusses the FPGA-implementation of the non-blocking cache design. Section V presents the evaluation of our design and compares it to a naïve non-blocking cache implementation. Section VI reviews related work, while Section VII summarizes our findings.

II. Background and Motivation

In Runahead execution an inorder processor exploits *memory level parallelism* (MLP) by prefetching additional memory blocks while there is an outstanding cache miss. Upon encountering a cache miss, the processor creates a checkpoint of its architectural state (e.g., registers) and

enters *runahead* execution mode. While waiting for the memory block to be retrieved, the processor continues executing subsequent independent instructions. Any result produced in this execution mode is later discarded. These intermediate results are solely used for generating additional cache misses and hence to prefetch memory data.

Upon delivery of the initial cache miss, the processor reverts back to the saved state effectively discarding all results produced during runahead execution. The processor resumes normal execution starting immediately after the instruction that missed. Any memory access that was initiated during runahead mode and brings useful data effectively prefetches this data and reduces execution time. The runahead references that bring useless data represent an overhead and may hurt performance. In practice runahead execution improves performance [2].

Runahead execution requires extending a conventional in-order processor with support for checkpointing, speculative execution under a miss, and the ability to issue multiple memory requests. Instructions executed during runahead mode require access to the data cache while a miss is pending. Therefore, the data cache must be non-blocking [3].

A conventional non-blocking cache uses Miss Status Handling Registers (MSHRs) to track outstanding misses [3]. MSHRs provide means of combining misses to the same cache line and of preserving ordering and thus cache data consistency while allowing multiple outstanding requests. Conventional MSHR implementations use a CAM-based structure and they are expensive to build on FPGAs both in terms of area and frequency.

Runahead execution offers an opportunity to revisit the conventional non-blocking design by tracking outstanding requests within the cache lines instead of in MSHRs. The following observations can be made:

1) Allowing multiple outstanding accesses to the same memory block during runahead mode has no advantage as the main purpose of the speculatively executing instructions is to overlap the retrieval of distinct memory blocks.
2) Section V demonstrates experimentally that Runahead execution is still effective even when multiple outstanding requests to memory blocks that map to the same cache line are not allowed.
3) Results produced in runahead mode are discarded and have no impact on the processor architectural state. Therefore, the processor has the option of simply discarding instructions corresponding to *secondary misses* (those described in (1) and (2)) without affecting correctness. Similarly, no ordering is required among requests send to memory during runahead execution.

This work presents an FPGA-friendly non-blocking cache that does away with MSHRs. Instead, outstanding misses are identified within the cache using a single *pending* bit stored along with each cache line. Whenever an address misses in the cache, the corresponding cache line is marked as *pending*. Subsequent accesses to this cache line would observe the *pending* bit and will be discarded. This simplifies cache implementation making it suitable for FPGAs.

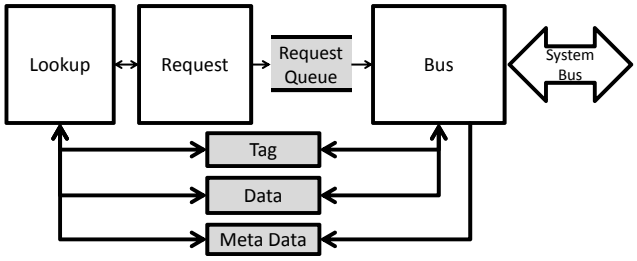

Figure 1. Non-blocking cache structure.

III. Non-Blocking Cache Architecture

This section discusses the non-blocking cache architecture and the optimizations applied to improve area and frequency. The cache is designed for single-cycle hits as hits are expected to be the common case. Cache misses and non-cacheable requests are handled using separate components which are triggered exclusively for such events and are off the critical path for hits. These events complete in multiple cycles.

Figure 1 depicts the basic structure of the non-blocking cache. The cache comprises a *Lookup*, a *Request*, and a *Bus* component. It also contains *Data*, *Tag*, *Request,* and *Metadata* storage units. The following subsections describe the function of each component.

A. Lookup

Lookup is the cache interface that communicates with the processor and receives load, store and non-cacheable requests. Lookup performs the following operations:

- For cache accesses, compares the request's address with the tag stored in the Tag storage to determine whether this is a hit or a miss.
- For cache hits, if this is 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. Other soft processor caches, such as those of Altera Nios II, use two cycles for stores [4]. In addition, the cache line is marked as dirty.
- For cache misses, Lookup marks the cache line as pending. Subsequent accesses to this line will be discarded (runahead mode) or blocked (normal mode) if pending bit is set.
- For non-cacheable requests and cache misses, Lookup triggers the Request component to generate appropriate

requests. In addition, queues the instruction metadata in the Metadata storage. Lookup blocks the processor interface until Request signals it has generated all the necessary requests.
- 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. In this mode all instructions execute speculatively and will be discarded. Accordingly, it is safe to discard such secondary misses. If the processor is in normal execution mode, Lookup stalls the processor.

B. Request

Request is normally idle waiting for a trigger from Lookup. When triggered, it issues appropriate requests directed at the Bus component by placing them in the Request Queue. Request performs the following operations:

- Waits in the idle state until triggered by Lookup.
- For cache accesses, Request generates a cache line read request. In addition if the evicted line is dirty, Request generates a cache line writeback request.
- For non-cacheable requests, depending on the operation, Request generates a read or write request.
- When all necessary requests are generated and queued, Request notifies Lookup and returns to the idle state.

C. Bus

The Bus component is responsible for servicing the 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:

1) Sender: The Sender sends requests to the system bus. Sender removes requests from the Request Queue and, depending on the request type, sends appropriate signals to the bus. A request can be of one of the following types:

- Cache Line Read: Read requests are sent to the 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.
- Cache Line Writeback: Write requests are sent to the bus for each data word of the cache line. Data words are retrieved from Data storage and sent to the system bus.
- Non-cacheable Read/Write: A single read/write request is sent to the peripheral through the system bus.

2) Receiver: This module handles the system bus responses. Depending on the processor's original request type, one of the following actions is performed:

- 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 to the processor the corresponding metadata from the Metadata storage. Receiver stores all the data words received in the Data storage. Upon receipt of the last word, Receiver stores the cache line tag in the corresponding entry in the Tag storage, sets the valid bit and clears both dirty and pending bits.
- 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.
- Load from Peripherals: Upon receipt of the data word, Receiver signals request completion to the processor and provides the data. The corresponding metadata is retrieved from the Metadata storage.

D. 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 storage units.

E. Request Queue

Request Queue is a FIFO memory holding requests generated by Request for Bus. Request Queue processes requests in the order they are generated.

F. Meta Data Queue

For outstanding requests, i.e., requests missing in the cache or non-cacheable operations, the cache stores the metadata accompanying the request, e.g., Program Counter, in the Metadata Queue. Eventually when the request is fulfilled, this information is provided to the processor along with the data loaded from memory or I/O. This information uniquely identifies the request. Metadata Queue is a FIFO memory and processes requests in the order they were received.

IV. FPGA Implementation

This section presents the non-blocking cache implementation. It discusses the design challenges and the optimizations applied to improve frequency and area.

A. Storage

Modern FPGAs contain dedicated block RAM (BRAM) storage units that are fast and take significantly less area compared to LUT-based storage. The rest of this subsection explains the design choices that made using BRAMs for most of the cache storage components possible.

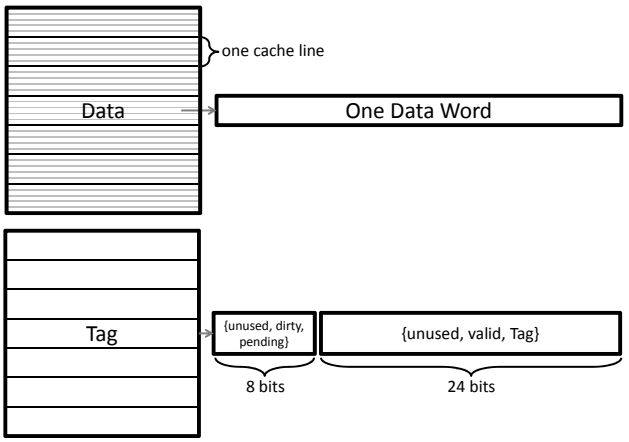

Figure 2. The organization of the Data and Tag storage units. A Cache line's data spans over multiple block ram entries. Each entry holds one data word. Cache line tags are stored along with valid, dirty and pending bits. Valid and Tag bits are stored in the lower 24 bits, while dirty and pending bits are stored in the higher eight bits. The rest of the bits are unused.

1) Data: Figure 2 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 [4] which supports byte, half-word, and word stores (one, two, and four bytes respectively). These are implemented using the BRAM *byte enable* signal [5]. Using this signal avoids two-stage writes (read-modify-write) which would also increase area due to necessary multiplexers.

2) Tag: Figure 2 depicts the Tag storage organization. Unlike cache line data, a tag fits in one BRAM entry. In order to reduce BRAM usage, valid, dirty and pending bits are stored with the tags.

Storing dirty and pending bits with the tags creates the following problem: 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. As Figure 2 shows, storing *valid* and *tag* bits in the lower 24 bits, and *dirty* and *pending* bits in the higher eight bits, make single-cycle writes possible using the *byte enable* signal. Using the *byte enable* signal, Lookup is able to change only the *dirty* and *pending* bits.

3) 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 [5].

As Figure 1 shows, both the Lookup and Bus components write and read to/from the Data and Tag storage. This requires four ports. Our design uses just 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. Despite Lookup and Bus both performing writes and reads to/from the data and tag storage, neither performs reads and writes at the same time.

For every Lookup access 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 reading the tag and other status bits from the Tag storage. Bus only writes to the Tag storage, when a cache line is retrieved from the main memory. Dedicating one address line to Lookup and one to Bus is sufficient to access the Tag storage.

For every Lookup access to the Data storage, Lookup either reads a word from or writes to a line. Bus may need to write to or read from the Data storage at the same time. This occurs if a writeback request is sent at the same time a cache line is received from memory. To avoid this conflict, Bus is restricted to sending a writeback data word only when the system bus is not delivering data. Forward progress is guaranteed as outstanding writeback 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. With this modification, dedicating one address line to Lookup and one to Bus is sufficient for accessing the Data storage.

B. State Machine Complexity

Conceptually, a cache comprises a CPU-side and a bus-side component. The CPU-side component looks up addresses in the cache, performs loads and stores, handles misses and non-cacheable operations, and sends necessary requests to the bus-side component. The bus-side component communicates with the main memory and system peripherals through the system bus.

Given the variety of operations that the CPU-side component handles, it requires a non-trivial state machine. The state machine uses numerous input signals and this reduces performance. The state machine also uses 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 large state machine leads to long critical paths and hence low operating frequency.

Higher operating frequency is possible by breaking the CPU-side component into two subcomponents, Lookup and Request, which cooperatively perform the same set of operations. This organization has its own disadvantages, however. In order for the two state machines to communicate, extra clock cycles are required for certain actions. Fortunately, these actions are misses and uncacheable requests which 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.

C. Latching the Address

We use BRAMs to store data and tags in the cache. As BRAMs are synchronous, the input address needs to be available just before appropriate edge (rising in our design) of the cycle when cache lookup occurs. In a pipelined processor, the address has to be forwarded to the cache from the previous pipeline stage, e.g., the execute stage in a typical 5-stage pipeline. After the first rising clock edge, the input address to the cache changes as it's forwarded from the previous pipeline stage. However, the input address is required for various operations, e.g., 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 the 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 generation. 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 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. In Section V we show that this technique combined with the rest of the optimizations improve area and frequency.

V. Evaluation

This section evaluates our non-blocking cache design. It first shows the potential performance advantage that Runahead execution has over an in-order processor using a non-blocking cache. It then reports area and frequency measurements for various non-blocking cache sizes using our design. It also compares our design to a naïve implementation of the non-blocking cache.

A. Methodology

In order to compare inorder, Runahead and OoO execution we use a cycle-accurate Nios II full system simulator capable of booting and running the uCLinux operating system [6]. The simulated processor models include a simple 5-stage pipeline, a Runahead processor, and OoO processor. The simulated cache resembles our optimized cache design. Table I details the simulated processor architecture.

Table I
Architectural properties of simulated processors.

I-Cache Size (Bytes)	32K
D-Cache Size (Bytes)	32K
Cache Line Size	32 Bytes
Cache Associativity	Direct Mapped
Memory Latency	20 Cycles
BPredictor Type	Bimodal
BPredictor Entries	512
BTB Entries	512
Pipeline Stages	5 (7 for OoO)
No. Outstanding Misses	2

We use benchmarks from the SPEC CPU 2006 suite which is typically used to evaluate desktop system performance [7]. We use these benchmarks as representative of applications that have unstructured ILP assuming that in the future soft-core based systems will be called upon to run demanding applications such as these. We use those benchmarks that would run without a floating-point unit. We use a set of reference inputs which are stored in main memory and accessed through the ramdisk driver. Measurements are taken for a sample of one billion instructions after skipping several billions of instructions so that execution is past initialization.

We implement the non-blocking cache in Verilog and use Quartus II v10.0 for synthesis and place-and-route on the Altera Stratix III EP3SL150F1152C2 FPGA. We compare our optimized cache implementation with a naïve cache implementation. The naïve implementation combines the two Lookup and Request components into a single component. It also performs two stage writes for changing dirty and pending bits. Finally, it latches the input address using the *next state* signal. We expect that even this straightforward non-blocking cache will be faster and more area efficient than a conventional, fully flexible non-blocking cache based on CAMs.

B. Runahead Execution

Figure 3 compares Runahead execution to a typical 5-stage in-order pipeline and a 1-way OoO processor. We use instructions per cycle (IPC) as a frequency independent metric for comparison. Runahead is able to achieve almost all of the benefits of OoO execution. On average, Runahead improves IPC by 14% over in-order execution.

C. Area

Figure 4 reports the number of ALUTs used by the optimized non-blocking cache for various capacities. For the optimized design, cache size has a negligible impact on ALUT usage. We also compare the optimized and naïve implementations. For large cache sizes, the naïve implementation uses slightly more area due to its higher complexity.

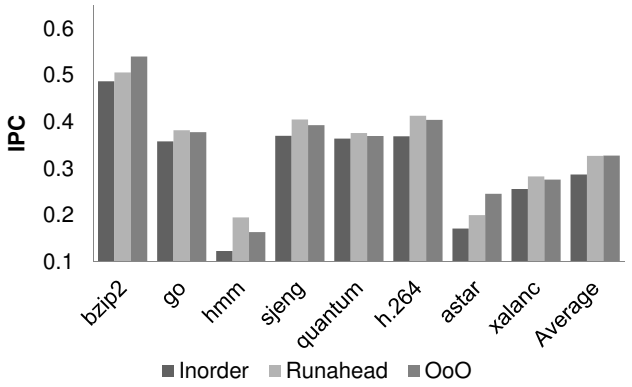

Figure 3. IPC comparison of inorder, runahead and 1-way OoO processors.

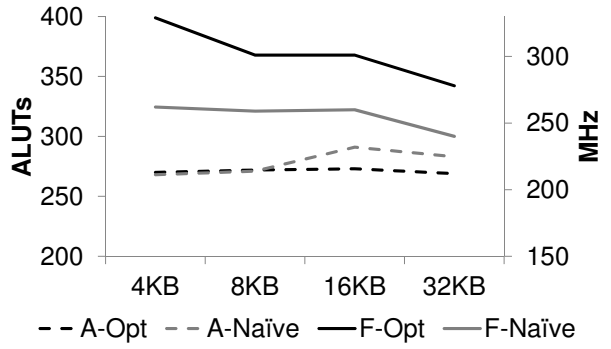

Figure 4. Area and frequency comparison of optimized and naïve non-blocking cache implementations. Solid and dashed lines show maximum frequency and area usage respectively.

The bulk of the cache is implemented using BRAMs. The vast majority of these BRAMs contain the cache's data, tag and status bits.

D. Frequency

Figure 4 reports the maximum clock frequency for various capacities of our non-blocking cache. Frequency decreases with cache capacity. Frequency decreases by at most 18% when capacity increases from 4KB to 32KB. The optimized implementation significantly outperforms the naïve implementation. The maximum difference is 25% between 4KB optimized and naïve caches which operate at 329MHz and 262MHz respectively.

VI. Related Work

Yiannacouras and Rose create an automatic cache generation tool for FPGAs [8]. 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. Coole et. al., present a traversal data cache framework [9]. Traversal caches are suitable for applications with pointer-based data structures. It is shown that, using traversal caches, for such applications a significant speedup, up to 27x, is possible. PowerPC 470S, a synthesizable implementation is available under a non-disclosure agreement from IBM. This core is equipped with non-blocking caches. A custom logic implementation of this core, PowerPC 467FP has been implemented by LSI and IBM.

VII. Conclusion

This work presented a highly efficient non-blocking data cache implementation for soft processors. A conventional non-blocking cache is expensive to build on FPGAs due to CAM based structures used in its design. Our non-blocking cache design exploits key properties of Runahead execution to avoid CAMs and instead stores information about pending requests in the cache itself. Additional optimizations have been described that improve frequency. A 4KB optimized non-blocking cache operates at 329MHz on Stratix III FPGAs while it uses only 270 logic elements. A 32KB cache operates at 278Mhz using 269 logic elements.

Acknowledgment

This work was supported by an NSERC Discovery grant and equipment donations from Altera. Kaveh 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 *19th Intl. Conf. on Field Programmable Logic and Applications (FPL)*, Prague, Czech Republic, September 2009.

[2] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *ICS '97: Proc. of the 11th intl. conf. on Supercomputing*. New York, NY, USA: ACM, 1997, pp. 68–75.

[3] S. Belayneh and D. R. Kaeli, "A discussion on non-blocking/lockup-free caches," *SIGARCH Comput. Archit. News*, vol. 24, no. 3, pp. 18–25, 1996.

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

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

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

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

[8] P. Yiannacouras and J. Rose, "A parameterized automatic cache generator for fpgas," in *Proc. Field-Programmable Technology (FPT*, 2003, pp. 324–327.

[9] G. Stitt, G. Chaudhari, and J. Coole, "Traversal caches: a first step towards fpga acceleration of pointer-based data structures," in *Proc. of the intl. conf. on Hardware/Software codesign and system synthesis*, New York, 2008, pp. 61–66.