An FPGA-based In-line Accelerator for Memcached

Maysam Lavasani, Hari Angepat, and Derek Chiou
Department of Electrical and Computer Engineering
The University of Texas at Austin
{maysamlavasani, hangepat, derek}@utexas.edu

Abstract—We present a method for accelerating server applications using a hybrid CPU+FPGA architecture and demonstrate its advantages by accelerating Memcached, a distributed key-value system. The accelerator, implemented on the FPGA fabric, processes request packets directly from the network, avoiding the CPU in most cases. The accelerator is created by profiling the application to determine the most commonly executed trace of basic blocks which are then extracted. Traces are executed speculatively within the FPGA. If the control flow exits the trace prematurely, the side effects of the computation are rolled back and the request packet is passed to the CPU. When compared to the best reported software numbers, the Memcached accelerator is 9.15x more energy efficient for common case requests.

1 Introduction

The performance of processors is now limited by their power consumption and thermal profile rather than the number of transistors [7], [9]. The demand for energy efficient computing has led to the rapid adoption of accelerators, from mobile SoCs with dedicated hardware for audio/video [25] to server processors with XML/Crypto accelerators [8].

While hardwired specialized accelerators are orders of magnitude more power efficient than CPUs [11], they are inflexible. To reduce the possibility the accelerator is rendered useless due to a bug or changing requirements, flexibility is often introduced, such as by adding support for parameters and/or loadable instructions or by implementing the accelerator in a field programmable gate array (FPGA) fabric. In either case, some of the gains of specialization are lost.

We propose a method to implement "in-line accelerators" on FPGA fabrics to accelerate server applications. An in-line accelerator sits between the network interface card (NIC) and the CPU and intercepts incoming packets going from the NIC to the CPU. In its general form, an in-line accelerator can (i) process the packets completely without CPU involvement, (ii) process the packets partially, leaving the rest of the computation for the CPU, or (iii) pass-through the packets to the CPU without processing them. For the context of this paper, we assume an architecture that supports only (i) and (iii), and not (ii). Thus, a server application is sliced into (i) a simple fast path that is executed by the in-line accelerator and (ii) a complex slow path that is executed by the CPU.

We evaluate our technique on Memcached. Full-system simulations of both the client and server systems show that a single in-line accelerator achieves 160% of the performance of a 2-way SMT Xeon core, while consuming 17% of its power. The synthesized accelerator on a medium-sized FPGA consumes only 6% of the FPGA resources. The accelerator handles more than 96% of requests on real workloads. We estimate that a CPU with the in-line accelerator is at least 2.3 times more energy efficient than a pure CPU solution. To the best of our knowledge, this is the first method to generate highly efficient in-line accelerators on FPGAs from application code at a reasonable programming effort.

Date of publication 15 Jul. 2013; date of current version 31 Dec. 2014.
Digital Object Identifier 10.1109/L-CA.2013.17

2 Background and Motivation

Routers are often split into a data plane (fast path) and control plane (slow path). The fast path which generally includes network processors, traffic managers, and switch fabrics accelerates the processing of common packets and leaves the processing of uncommon packets (control packets) to general purpose processors.

Many network processors are multi-core ASIPs (Application Specific Instruction Processors) with specialization at different levels, including specialized instruction supplies, customized datapaths, and highly tuned interconnects, memory systems, and IO interfaces. They are much higher in throughput and energy efficiency comparing to CPUs for a wide range of networking applications, from high-end routers to edge routers and firewalls.

There have been many attempts to manually split applications into fast path and slow path components [4], [23]. On the other hand, other research work [20], [24] manually implemented server applications on FPGA resources. Our focus is to create a general technique to create FPGA-based in-line accelerators. The main requirement for this generalization is a method to automatically create a fast path and to implement the fast path on a highly efficient micro-architecture. Emerging FPGA+CPU platforms ([6], [15], [26]) make this technique attractive.

A natural approach to automatically generate the fast path is to detect hot traces, consisting of multiple basic blocks and then slice the application to extract those hot traces. This approach has been used to accelerate desktop applications in architectures with tightly coupled cores and accelerators with a shared register file [10]. However, in-line accelerators can further improve the performance by implementing entire hot traces in the accelerator inside the NIC. Doing so enables the accelerator to process most packets without CPU involvement. In addition, the CPU cores require no special modifications.

We used Valgrind [22] to profile Memcached servicing one million *get* requests. Figure 1 shows (i) the distribution of dynamic instructions across static ones and (ii) the average number of dynamic instances of each instruction per Memcached request. For most of the static instructions the number of dynamic instances per request is close to zero. These numbers lead us to determine the percentage of requests that can be serviced in a hot trace. We found

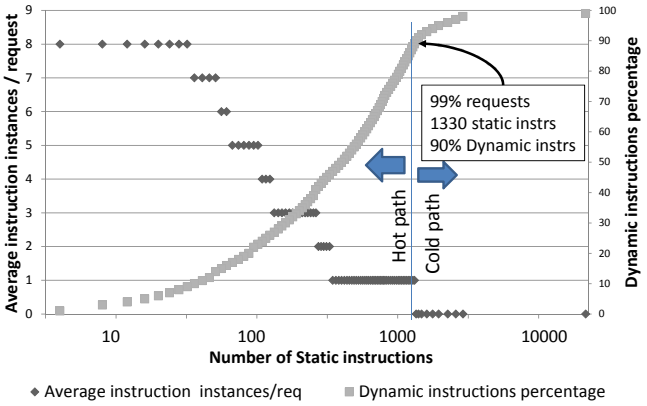

Fig. 1. Memcached dynamic/static instruction patterns. Each static instruction might get executed multiple times and creates multiple dynamic instructions.

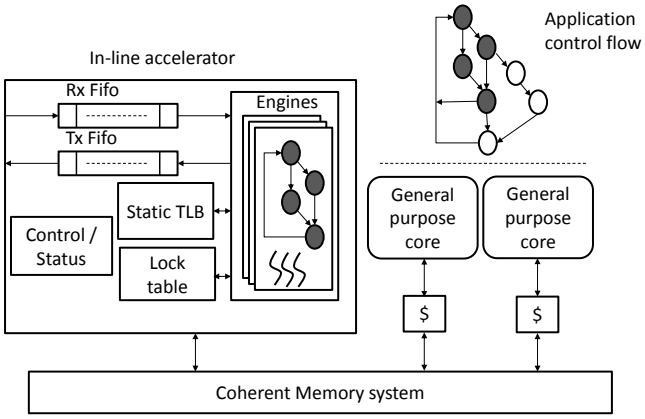

Fig. 2. Proposed In-line accelerator architecture. Shaded circles represent hot control flow blocks that are used to service most request packets, while unshaded circles represent the remaining control blocks

that 99% of *get* requests can be handled using a single hot trace consisting of 1330 static instructions. This pattern is the principle motivation to create an accelerator by covering the hot instruction traces using efficient hardware and leave the processing of the remaining requests which jumps out of the hot traces to the CPU.

3 In-line Accelerators

In a conventional server architecture, the NIC's main functionality is to copy the incoming and outgoing packets to/from the memory system. In a conventional server application, the CPU expects packets to be available in the memory after which it will (i) parse the request packets and extract the relevant fields that form the arguments for the request, (ii) process the request by performing computation and potentially modifying global data structures, and finally (iii) create response packet(s) if required. If the application lives in the user space, additional overhead is imposed to copy the packet data across various buffers/privilege-levels including NIC FIFO, kernel network stack, application level buffers. While zero-copy and user-space networking [13], [21] can be used to minimize the user/kernel distinction at the cost of blurring the protection/privilege separation, such approaches still maintain a strong distinction between the NIC buffer and the processor memory.

An in-line accelerator architecture (Figure 2) redraws how computation is done by combining the NIC and the in-line accelerator which (i) receives the incoming request packets, (ii) processes the request packets by accessing and modifying global application data-structures through a coherent port without involving the CPU, and finally (iii) sends the response packets if required. The in-line accelerator processes packets speculatively, assuming the packet is a common case the accelerator can handle. If the accelerator determines it cannot handle the packet, it "bails out" from the fast path by rolling back what the accelerator did speculatively, and passes the request packet to the CPU cores via the conventional NIC interface.

3.1 Accelerator Micro-Architecture

In previous work, we developed an architecture and a compiler to synthesize an FPGA-based Layer 3 packet processor [16]. We use the same micro-architecture for in-line acceleration. The micro-architecture consists of multiple multi-threaded engines. The independent input packets are distributed among different engines for processing.

Each engine is a multi-threaded finite state machine with an application specific datapath. Several instructions are fused at compile-time to form a single finite state. In each state, engines can perform chained arithmetic/logic operations, predicated operations, and access to off-engine resources. Each state, however, accesses the off-engine resources, including packet memory, or any shared data between engines, no more than once. Although the access to off-engine resources might have variable delay, the engine infrastructure (i) guarantees the termination of operations of a single state in a lock-step manner and (ii) switches to another execution thread, processing another packet, whenever access to off-engine resources incur long latency.

The focus of this paper is to couple such an FPGA-based accelerator with a CPU to handle more complex protocols and applications. Using a shared memory, data structures can be shared between the fast path running on the accelerator and the slow path running on the CPU. A kernel driver allows a user application to allocate structures from a shared pool of physical memory, which is accessible from the accelerator. The driver also informs the accelerator of this mapping, which is stored in an address translation table on the accelerator. The accelerator uses this table to translate virtual addresses when manipulating the pointers. In addition, the accelerator contains a hardware lock engine that is also used by CPU threads to guarantee mutual exclusion.

3.2 Generating Accelerators

There are five steps to generate fast path in-line accelerators: (i) profiling the application to detect the hot basic blocks, (ii) slicing the application into fast path and slow path based on the result of profiling, (iii) writing the bail out code, (iv) refactoring the code into synthesizable processing steps [16], and (v) synthesizing the fast path code into hardware.

The automation of the above process is out of the scope of this paper. However, it is worth to mention that the steps (i), (ii), and (v) are fully automated in our system. We are currently working on automating step (iv) using high-level synthesis techniques. Automating step (iii) is future work.

However, server applications usually isolate updates associated with a client request until the entire processing of the request is completed. In such cases, the bail out code to roll back the side effects of the isolated computation is simple. Memcached *get* operation, for example, isolates the requested object from potential concurrent writers, making it straightforward to write the bail out code.

In addition, mechanisms similar to hardware transactional memory can be leveraged to roll back automatically. Unlike conventional transactional memory programs where rollbacks are triggered by conflicting memory accesses, in-line accelerator rollbacks would be self-triggered to bail out of the fast path.

4 Accelerating Memcached

Memcached [19] is an open source key-value system for in-memory objects which is often used as an application level cache for conventional data repository systems [18]. Distributed Memcached clients use a consistent hashing scheme to find the home server of an object of interest. The server caches frequently requested objects in a hash table and uses its own hash function to find the object. Memcached has a number of commands for manipulating the objects (e.g. *get*, *set*, *delete*, etc.) with a nearly 30-to-1 ratio of *get* to other types of requests in real workloads [1].

Recently, researchers in [5] implemented a subset of Memcached commands on an FPGA. Comparing to CPU-based solutions, they showed a significant improvement of energy efficiency. Only two main Memcached commands — *get* and *set* — were implemented on top of the UDP protocol in order to fit the design in the target FPGA.

Memcached consists of 14 major commands. Due to the requirement for a reliable communication channel, all commands except *get* should be implemented on top of TCP protocol. Such compromise re-emphasizes the fact that for some complex applications, it is difficult for an FPGA-only solution to be complete.

4.1 Memcached In-line accelerator

Profiling Memcached using a mixed *get*/*set* workload and slicing the source code results in a fast path for common *get* requests and leaves the handling of remaining requests (a small portion of *gets* and all other commands i.e. *sets*, *deletes*, etc.) to the CPU. The code that processes the *get* request, including request parsing, item hash calculation, item look up, LRU update, and response assembly, are all in the fast path. When a cached object has an expired time stamp, the request is handed to the slow path which kicks off relatively complex slab management. We add the bail out code, less than 30 lines in the original Memcached, to the fast path code in order to safely abort a request and to handover a request to the cold path software threads. Table 1 specifies the details on the Memcached fast path breakdown. The breakdown does not include the hot instructions from dynamic libraries.

4.2 Evaluation

The proposed architecture is evaluated using the gem5 [3] simulator, modeling an Alpha DEC Tsunami system. A two-node system is configured with mcblaster (a standard Memcached load generator) running on the client and the Memcached daemon running on the server. The client and the server are connected using a direct simulated Ethernet link.

TABLE 1
The breakdown of the fast path hot trace in Memcached in terms of LOCs (Lines Of Code) and hot static instructions

Function	Caller function	LOCs	Hot LOCs	Hot instrs
drive_machine	-	320	82	261
try_read_cmd	drive_machine	129	33	83
add_msg_hdr	try_read_cmd	36	23	41
process_get_cmd	try_read_cmd	168	37	126
tokenize_cmd	try_read_cmd	42	36	138
item_update	process_get_cmd	8	5	24
item_get	process_get_cmd	9	5	26
item_lock	item_update/item_get	3	3	36
item_unlock	item_update/item_get	3	3	15
do_item_get	item_get	49	14	46
do_item_update	item_update	13	4	8
hash	item_update/item_get	122	13	180
assoc_find	do_item_get	24	6	37
item_link_q	do_item_update	21	10	20
item_unlink_q	do_item_update	16	10	20

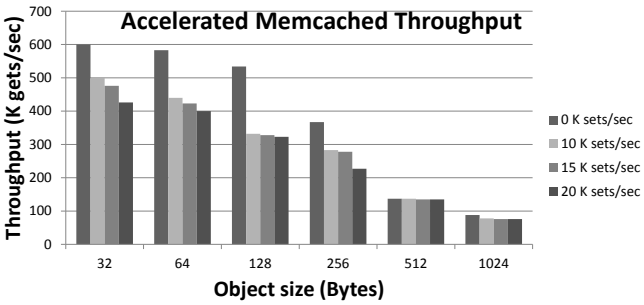

Fig. 3. Throughput of accelerated Memcached with single engine accelerator

On the server system, the Alpha core acts as the slow path core. The baseline NIC model contains a state machine that copies incoming packets into the system memory and outgoing packets from the system memory, both through the DMA port. The NIC state machine is modified such that upon receiving a new packet, the fast path state machine processes the packet. If the fast path is capable of handling the request, the steps are executed as part of the NIC state machine. Otherwise, NIC proceeds with its normal functionality, which is to copy the packet to the system memory. The modified NIC is simulated to run at 100MHz.

Memcached is populated with half a million objects and then warmed up for 5 seconds of target execution time in *atomic* mode. After the warm up phase, simulation mode is changed to *timing* for measurements and run for 2 seconds of target execution time. Figure 3 demonstrates Memcached throughput using a single-engine FPGA-based in-line accelerator. We vary the fraction of *set* traffic that is randomly inserted into the workload. As expected, performance degrades with the number of *set* operations. However, as we move to increasing object sizes, the impact of the *set* slow path serialization is minimized as more time is spent outside of the critical section to service each request.

We synthesize the generated in-line accelerator to a Xilinx Virtex-5 TX240T FPGA part, as a measure of area and power consumption for the FPGA fabric portion of CPU+FPGA architecture. The accelerator consumes 9563 (6%) of the FPGA's LUTs and 5808 (1%) of the FPGA's registers. Based on the performance measurement of our accelerated system,

TABLE 2
Estimation of accelerated Memcached power efficiency using hybrid FPGA+CPU architecture.

	General-purpose dual-thread cores	Accelerator dual-thread engines	Cores power (watts)	Uncore power (watts)	Accelerator power (watts)	Performance (requests/sec)	Power efficiency (requests/sec/watt)
CPU	8	0	96	44	0	$<$ 3150K	$<$ 22.5K
CPU+FPGA	1	4	12	44	8	3200K	51.61K

a single threaded accelerator engine is capable of 583K $gets/sec$ while consuming less than 2 watts of power for 64 bytes objects. The targeted FPGA can accommodate 12 dual-thread accelerator engines at a very reasonable 72% occupancy. Assuming there are no other limitations in the system, that single, medium-sized FPGA can deliver almost 14M requests/sec.

We use the recently reported numbers of 350K $gets/sec$ for a Xeon core as our baseline [14]. McPat [17] is used to estimate the breakdown of power consumption of the CPU used in [14]. Xilinix's xpower tool was used to estimate the FPGA's power consumption. When comparing only core power versus only FPGA power, our accelerator is 9.15 times more power efficient than the Xeon core solution.

We also estimate the energy efficiency of the whole CPU+FPGA accelerated solution and compare it with CPU-only solution. The power efficiency of an eight core Xeon processor with configuration reported in [14] is 22.5K requests/sec/watt for get-only workload. Assuming that the same chip is equipped with an FPGA fabric that can accommodate an accelerator with four dual-thread engines, the estimated chip power efficiency will be 51.61K requests/sec/watt. Table 2 shows the power, throughput, and energy efficiency for (i) the base-line Memcached server and (ii) the in-line accelerated Memcached server. We reported the get-only performance as a ceiling for the non-accelerated mixed-load performance. The CPU+FPGA performance, however, is based on a mixed workload with 30 $gets$ to 1 set ratio. Our proposed solution also outperforms other recent many-core [2] and GPU-based [12] solutions.

5 Conclusions and Future Work

The notion of slicing a networking application into the fast path and the slow path has been used in many communication and networking systems. This paper proposes to apply such techniques to server applications amenable to speculatively executing a hot path by slicing the application and generating the fast path hardware accelerator. In addition to the Memcached application, our preliminary results from slicing a user space TCP protocol stack, as well as a light-weight web server confirms the existence of fast paths in such applications. The heart of the acceleration technique is a speculative execution model that enables the in-line accelerator to perform computation on behalf of conventional cores, while still being able to fall back to software for the complex slow path.

The in-line accelerator architecture also offers significant benefits to latency sensitive applications by stripping various protection/privilege and artificial boundaries imposed by existing interfaces, and replacing them with the fast path accelerators. Since some applications [21] are highly sensitive to latency, this approach offers benefits beyond performance and energy improvements. We plan to apply our FPGA-based in-line acceleration technique to latency sensitive applications such as financial applications. Exploring the trade-offs between FPGA resources and modern network processors as the substrate for in-line acceleration is another interesting future work.

References

[1] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS/PERFORMANCE*, 2012.
[2] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. In *IGCC*, 2011.
[3] N. Binkert, B. Beckmann, G. Black, K. Reinhardt, A. Saidi, A. Basu, J. Hestness, R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. Hill, and D. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 2011.
[4] G. Brebner. Single-chip gigabit mixed-version ip router on virtex-ii pro. In *Field-Programmable Custom Computing Machines*, 2002.
[5] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala. An fpga memcached appliance. In *FPGA*, 2013.
[6] Convey Computers.
[7] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. *SIGARCH Comput. Archit. News*, 2011.
[8] H. Franke, J. Xenidis, C. Basso, B. Bass, S. Woodward, J. Brown, and C. Johnson. Introduction to the wire-speed processor and architecture. *IBM Journal of Research and Development*, 2010.
[9] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The greendroid mobile application processor: An architecture for silicon's dark future. *Micro, IEEE*, 2011.
[10] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *MICRO*, 2011.
[11] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA*, 2010.
[12] T. Hetherington, T. Rogers, L. Hsu, M. O'Connor, and T. Aamodt. Characterizing and evaluating a key-value store application on heterogeneous cpu-gpu systems. In *ISPASS*, 2012.
[13] Packet processing on Intel architecture.
[14] Enhancing the Scalability of Memcached. http://software.intel.com/en-us/articles/enhancing-the-scalability-of-memcached.
[15] Intel Quick Assist Platform.
[16] M. Lavasani, L. Dennison, and D. Chiou. Compiling High Throughput Network Processors. In *FPGA*, 2012.
[17] S. Li, H. Ahn, D. Strong, B. Brockman, M. Tullsen, and P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, 2009.
[18] Scaling Memcached at Facebook.
[19] Memcached Wiki Page.
[20] S. Muhlbach and A. Koch. A dynamically reconfigured network platform for high-speed malware collection. In *ReConFig*, 2010.
[21] Open onload project.
[22] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *ATEC*. USENIX Association, 2005.
[23] L. Shalev, V. Makhervaks, Z. Machulsky, G. Biran, J. Satran, M. Ben-Yehuda, and I. Shimony. Loosely coupled tcp acceleration architecture. In *HOTI*, 2006.
[24] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang. Fpmr: Mapreduce framework on fpga. In *FPGA*, 2010.
[25] Tegra 2 and Tegra 3 Super Processor.
[26] Xilinx Zynq-7000 SoC platform.