FlexCache: Field Extensible Cache Controller Architecture Using On-Chip Reconfigurable Fabric

Daniel Lo, Greg Malysa, G. Edward Suh
Cornell University,
Ithaca, NY, USA
{dl575, gjm76, gs272}@cornell.edu

***Abstract*—In today's microprocessors, the cache architecture is highly optimized for one particular design and cannot be changed after fabrication. While allowing efficient implementations in dedicated logic, this inflexibility also implies that new techniques cannot be deployed in the field. This paper presents FlexCache, a flexible cache architecture that uses on-chip reconfigurable fabric to enable new extensions to be added in the field after fabrication. We evaluate the flexibility and efficiency of the architecture through an RTL prototype implementation of the cache along with example extensions such as cache performance counters, side-channel protection, prefetching, various replacement policies and computation acceleration. The results show that various types of extensions can be realized on FlexCache with minimal impact on performance, power, and area.**

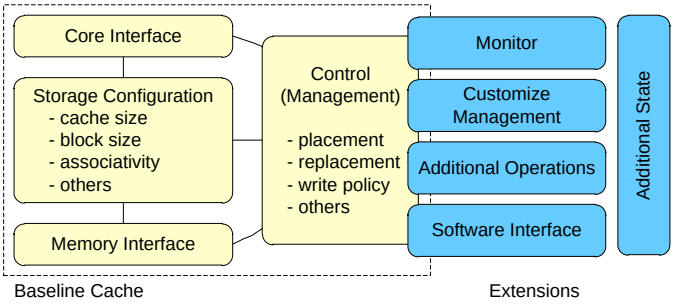

Fig. 1. Types of cache extensions supported by the FlexCache architecture. Light (yellow) blocks represent baseline cache functions and dark (blue) blocks represent FlexCache extensions.

I. Introduction

The memory hierarchy plays a central role in modern microprocessors. Researchers have proposed numerous ideas and techniques that augment caches to improve performance, enhance security and reliability, and extend functionality [1], [2], [3]. However, today's cache implementation is typically fixed and optimized for a particular cache design. This enables the cache implementation to be highly efficient, yet at the same time means that even a small change in the cache requires re-design and re-fabrication of a microprocessor. Such a hardware change is often prohibitively expensive because of ever increasing NRE (Non-Recurring Engineering) costs.

In this paper, we propose a flexible cache controller architecture, named FlexCache, that utilizes reconfigurable fabric to enable field reconfigurability of cache functions. The FlexCache architecture aims to enable a broad range of cache extensions to be added or updated in the field after fabrication. Cache implementations can be updated or patched like software which can be especially important for security extensions. The flexibility can also be used as a way to experiment with new mechanisms on real systems before being incorporated in future microprocessor designs.

The main insight in the proposed architecture is that many cache extensions can be realized by either transparently monitoring cache operations or changing infrequent operations. This observation allows our architecture to provide flexibility without significantly impacting common cache accesses. Under most circumstances, FlexCache allows dedicated cache logic to perform its operations and monitors them in the background, effectively incurring no overheads. The architecture utilizes the reconfigurable fabric to change cache operations only when needed.

The main contributions of this paper include identifying the cache operations that should be monitored, determining a minimal set of primitive operations for cache control customization, and an architecture design using these to create a flexible cache controller. The resulting cache controller allows customization of its operation while maintaining efficiency for common cache operations. The contributions also include implementing an RTL prototype which is used for evaluation.

This paper is organized as follows. Section II describes the FlexCache architecture and Section III shows how a set of example extensions can be realized on the FlexCache architecture. Section IV presents evaluation results from the RTL implementations. Section V discusses related work and Section VI concludes the paper.

II. FlexCache Architecture Design

A cache architecture can be made completely flexible if all of its structures and operations are made programmable. However, there exists an inherent trade-off between flexibility and efficiency. A fully programmable cache design will inevitably slow down common operations and result in poor overall performance. For example, an FPGA fabric can implement almost arbitrary cache designs, but it is significantly less efficient than custom ASICs.

In FlexCache, we aim to provide a flexible cache architecture that can be extended and customized after fabrication, without noticeably impacting the baseline cache performance when no extension is added. In this context, the FlexCache

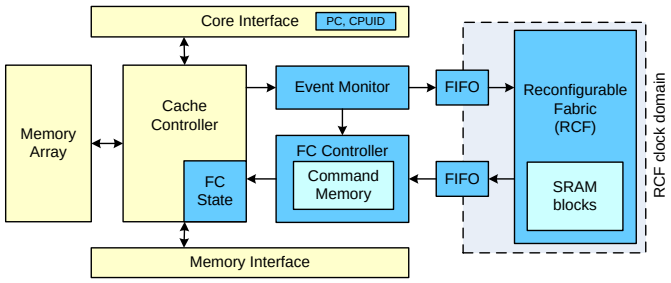

Fig. 2. Overall FlexCache (FC) architecture. Light (yellow) boxes represent the baseline cache and dark (blue) boxes represent the changes for FlexCache.

architecture is mainly designed for two types of cache extensions: adding transparent operations that can be performed in parallel to normal cache functions and customizing infrequent operations such as cache misses. Many cache extensions for security, reliability, verification, and even performance optimizations fall into these categories.

Fig. 1 shows the ways in which FlexCache extends cache functionality. FlexCache enables transparent monitoring of cache events, customizing cache management, and adding new operations. It also includes a software interface and additional state that extensions can leverage. It does not aim to support changing a physical cache configuration such as cache size, block size, associativity, etc. As a result, common datapaths do not need to be reconfigurable and typical cache operations can remain efficient.

A. Architecture Overview

The high-level block diagram of the FlexCache cache architecture is shown in Fig. 2. The baseline cache consists of a controller, memory arrays for tags and data, and interfaces to a processor and a lower-level memory. The cache controller contains a finite state machine (FSM) that orchestrates the overall cache operation by generating control signals on each cache event.

The cache controller in FlexCache handles memory operations as a normal controller would while providing optional capabilities to monitor and customize handling of a cache event. To support transparent monitoring of cache events, an event monitor block is attached to a cache controller. The monitor sends a message with details of an event to a reconfigurable fabric, called a FlexCache fabric, if the event is configured to be monitored. The FlexCache fabric performs bookkeeping and computation. In our current design, events can be configured to be monitored or not based on the type such as read hits, write hits, read misses, and write misses as well as an address range.

To support customization of each event, the state machine in the cache controller is augmented with a special state, named a FlexCache state, which takes commands from a programmable controller (FlexCache controller). On a cache event, the cache controller's FSM can be configured to transition to the FlexCache state. The cache controller can be configured to transition to this state before or after normal handling of the cache event. Transitioning before normal handling allows custom handling of the event. Transitioning after allows normal handling of the event with no overhead while allowing FlexCache to add additional operations afterwards. The FlexCache state supports a set of primitive operations that can be used to build cache functions, such as handling a miss. The FlexCache controller determines which primitive operations need to be performed on each event based on a sequence of commands that are stored in its command memory along with computation results and meta-data from the FlexCache fabric. The FlexCache controller can also request the main controller to enter the FlexCache state explicitly by sending a command. When done, the FlexCache controller releases its control and the cache controller returns to its normal operation.

The FlexCache fabric communicates with the event monitor and the controller through two FIFOs, one FIFO in each direction. The use of FIFOs allows the FlexCache fabric to operate in a different clock domain from the cache if necessary. FIFOs also decouple the fabric from the controller by allowing multiple messages to be queued up. This decoupling is essential for hiding communication latencies and enabling the FlexCache fabric and other modules to operate in parallel.

B. Event Monitor

The event monitor sends a message to the FlexCache fabric and the FlexCache controller on cache events that are configured to be monitored. These messages contain the event type such as read hit, read miss, write hit, write miss, etc. along with additional information on the details of the event. In our current design, the detailed information includes the instruction address (PC), the memory address of the access, the data value (only for a write operation), the cache set number in a set-associative cache, the privilege level, and the processor ID (only for a shared cache). In most cases, the monitoring messages do not affect the main cache controller operation. However, if the FIFO is full when a message needs to be sent, the controller will be stalled.

C. FlexCache Controller

When the cache controller is in the FlexCache state, it waits for a command from the FlexCache controller to determine its operation. Table I summarizes the commands that are supported by the FlexCache architecture. To reduce unnecessary data transfers between the main controller and the FlexCache side, the main controller maintains a data register that keeps the current working copy of a cache block. The FlexCache commands operate on this special register in a way that is similar to an accumulator style instruction set.

The commands can be put into four categories: data movement between the controller and the FlexCache fabric, read/write cache memory arrays, read/write a lower-level memory system, and the interface to the processing core. The commands are designed to represent primitive operations that can be combined to handle cache events. For example, a cache miss operation can be implemented by executing `cinst_readMem`, `cinst_returnProc`, and `cinst_writeCache`. While not shown in the table, each

TABLE I
FLEXCACHE COMMANDS FROM THE FLEXCACHE CONTROLLER TO THE CACHE CONTROLLER.

Command	Arguments	Description
`cinst_readFromFabric`	`data`	Put data from the FlexCache fabric into the data register in the cache controller.
`cinst_writeToFabric`		Send the data register value from the cache controller to the FlexCache fabric.
`cinst_readCache`	`index, set`	Read the cache memory arrays. Put the values in the cache controller data register.
`cinst_writeCache`	`index, set`	Write the cache controller data register values into the cache memory arrays.
`cinst_readMem`	`addr`	Read data block `addr` from the lower-level memory into the data register.
`cinst_writeMem`	`addr`	Write the cache controller data register to `addr` in the lower-level memory.
`cinst_returnProc`	`offset`	Return a word from the cache controller data register to the processor.

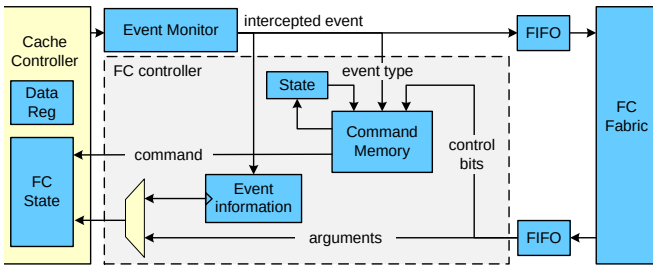

Fig. 3. FlexCache (FC) controller block diagram. The controller implements a state machine to provide FlexCache commands in the FlexCache state.

command has a special "done" bit which returns the controller FSM to the idle state after the command. Another flag which exists is an "unstall" bit which allows the processor to resume execution while keeping the cache controller in the FlexCache state to continue operating in the background.

Fig. 3 shows the block diagram of the FlexCache controller. The FlexCache controller creates a sequence of commands on each intercepted cache event by combining information from its command memory module and the FlexCache fabric. The command memory stores a sequence of FlexCache commands that need to be executed on a cache event, and effectively implements a programmable state machine. The memory is addressed using a concatenation of the current state, a cache event, and control bits from the FlexCache fabric. Each entry in the memory contains a FlexCache command opcode, an argument select bit, and the next state. The argument select bit specifies whether command arguments are from a register that stores the information of the current event or from the FlexCache fabric. The information from the current event, such as address and data, are often the correct arguments for the command. In this way, the FlexCache fabric is only used when the arguments need to be customized.

D. Core and FlexCache Fabric Communication

In many cases, a software program on the processing core needs to communicate with an extension on the FlexCache fabric either by reading or writing a value. For example, a write to the FlexCache fabric can be used to control an extension and a read operation can be used to obtain a result from a performance counter or an in-cache computation. While there are multiple ways to implement such a communication channel, our architecture currently treats the FlexCache fabric as a memory-mapped device. This approach is implemented by simply configuring the corresponding address range to be monitored. A write to the designated address is treated as a monitored event that generates a message to the FlexCache fabric. A read from the address is treated as an intercepted event that is handled in the FlexCache state where the FlexCache fabric can send a value back to the processor. Reads from the FlexCache fabric can be blocking and serve as a synchronization primitive between the core and the FlexCache fabric.

E. FlexCache Fabric Architecture

The high-level FlexCache architecture is independent from the micro-architecture of the FlexCache fabric and is applicable to various types of reconfigurable fabrics. In this paper, we use a standard LUT-based FPGA architecture, which includes standard Configurable Logic Blocks (CLBs) with LUTs and flip-flops. The FPGA fabric is chosen because it is general enough to support any type of computation while being efficient for bit-level operations that the cache extensions typically perform to generate command arguments and condition bits. In addition to standard CLBs, the FlexCache fabric also includes SRAM blocks similar to the ones in modern FPGAs in order to efficiently store meta-data. Further optimization of the FlexCache fabric remains as future work.

F. Advanced Cache Organizations

Non-Blocking Caches: Modern microprocessors with multi-threading or dynamic scheduling capabilities often use a non-blocking cache which allows a new access to be serviced under a cache miss by decoupling operations on a cache miss using buffers [1]. On a cache miss, instead of handling the entire miss, the controller stores information on the miss in Miss Status Holding Registers (MSHRs) and an address stack, initiates an access to the lower-level memory, and returns to service other cache accesses. When the requested block returns from the memory, the cache looks up the MSHRs and the address stack to complete the miss. To be applied to the non-blocking cache, the FlexCache architecture also needs to decouple a cache miss into two events: the cache miss initiation and the data return. Then, the two events can be monitored and intercepted separately. As an example, to customize a cache replacement policy, only the data return event needs to be intercepted. In addition, the FlexCache controller also needs to be able to issue commands for controlling the MSHRs.

Cache Coherence: In multi-core systems, caches need to incorporate a cache coherence mechanism. From the FlexCache's perspective, cache coherence introduces a new set of

cache events (snoop accesses) that happen largely in parallel to cache accesses from the core. The FlexCache architecture can be used to monitor or customize coherence events in the same way that it handles regular cache accesses. The main change is that the architecture needs an extra set of FlexCache modules, including event monitor, FlexCache controller, and FIFOs, so that the coherence events can be handled in parallel to other cache events. The FlexCache fabric can be shared between the two sets of FlexCache modules.

Multi-Level Caches: The FlexCache architecture is not specific to the L1 cache and is applicable to any level of cache memory. For example, the FlexCache architecture for an L2 cache will handle requests from the L1 cache and access the main memory (or an L3 cache) as the next level. The overall architecture and the FlexCache commands remain exactly the same except that the `cinst_returnProc` becomes `cinst_returnL1` that returns an L1 cache block instead of a word.

III. Example Extensions

This section shows how the FlexCache architecture can be used to implement cache extensions using five representative examples: cache event counters, Partition-Locked cache, prefetching, custom replacement policy, and AES acceleration. The descriptions here are based on a cache with a write-through no-allocate write policy.

Cache Event Counters: Performance counters including counters for cache events can be used to understand application characteristics and optimize performance in various ways such as re-compilation, intelligent operating system scheduling, etc. The FlexCache architecture enables custom cache event counters to be added after fabrication. For example, counters may estimate the number of misses for various cache sizes [2] or even be tailored to an application and monitor cache performance only for specific memory objects.

To implement performance counters, we configure the event monitor to send messages to the FlexCache fabric on all relevant cache events without transitioning to the FlexCache state. In our case, this includes read misses, read hits, write misses, and write hits. The FlexCache fabric maintains a set of counters and increments the corresponding counter based on the type and the address of a cache event. The counter values can be cleared or read using the memory mapped accesses from the processor. The monitoring extension performs its operation completely in the background and does not impact normal cache accesses as long as the FlexCache fabric can keep up with the cache accesses.

Partition-Locked Cache (PLcache): Recent studies have found that a shared cache may allow sensitive information such as a secret key to leak from one program to another through a timing side-channel. FlexCache allows such hardware bugs to be fixed. Fundamentally, the cache side-channel attacks come from the interference between two programs. Therefore, one way to protect against the cache-based side-channel attacks is to partition the cache between programs. We implemented one such proposal named a Partition-Locked cache (PLcache)

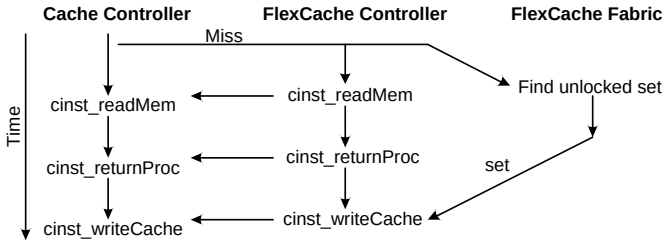

Fig. 4. Diagram of PLcache extension operation.

[3]. PLcache allows cache lines to be locked by a sensitive program, preventing an attacker from evicting the protected cache lines. Cache lines are locked and unlocked with special instructions. The operating system is in charge of enforcing the security policy for locking and unlocking lines.

In order to implement PLcache on FlexCache, we configure the cache to monitor and intercept every miss that requires replacing a cache block. Thus, on a miss, the cache controller transitions to the FlexCache state and sends a miss event message. The handling of this event is shown in Fig. 4. On a miss, the FlexCache controller's command memory is configured to first send a `cinst_readMem` command using the same address that the miss occurred at. Second, the command memory will indicate for the cache controller to return the data to the processor using a `cinst_returnProc` command while also indicating that the processor can be unstalled. While these first two commands are being processed, the FlexCache fabric selects an unlocked block to evict from the cache and sends the set number to the FlexCache controller. There is a delay in communicating with the FlexCache fabric because of the FIFOs. The FlexCache fabric maintains a lock bit for each cache block in its SRAM blocks. In our implementation, locking and unlocking of a cache block is done through the memory-mapped communication. In the final step, the FlexCache controller issues a `cinst_writeCache` command using the arguments from the FlexCache fabric to indicate which cache block to replace. If all cache blocks were locked, then the control bits sent by the FlexCache fabric would indicate to the FlexCache controller to skip this step. This final command is sent with the done bit set so that the cache controller exits the FlexCache states and resumes normal operation.

Prefetching: Prefetching is a popular performance optimization technique that tries to predict future memory accesses and bring in the data before they are requested by a program. FlexCache allows intelligent prefetching, potentially customized for specific applications. As a proof of concept to show the flexibility of our architecture, we implemented a stride prefetcher. For each memory access, the stride prefetcher compares the address accessed with the address accessed the previous time that instruction was executed. From this, the prefetcher calculates the stride of the access. If it sees the same stride several times, then the prefetcher infers that the program may continue to follow this pattern of stride accesses and will prefetch the next memory address in the sequence.

To implement this on FlexCache, the cache is configured to

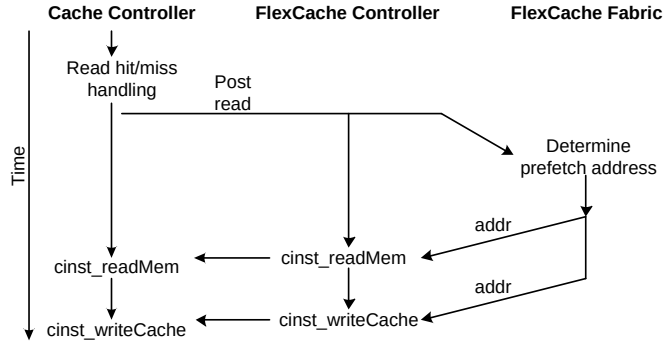

Fig. 5. Diagram of prefetching extension operation.

monitor read misses and hits. The FlexCache fabric monitors these events and stores meta-data information, such as the address of the access and stride, in the SRAM blocks. The cache controller is configured to transition to the FlexCache state after the normal handling of a read. After normal handling of a read, the FlexCache fabric decides whether to perform a stride prefetch based on the stored meta-data and the new access information. If a stride access pattern is detected, then the FlexCache fabric issues control bits to the FlexCache controller indicating to start the prefetch sequence and the address from which to prefetch. The handling of this case is shown in Fig. 5. The FlexCache controller will issue a `cinst_readMem` followed by a `cinst_writeCache` using the prefetch address specified by the FlexCache fabric before exiting the FlexCache state. If no stride access pattern is detected, then the FlexCache fabric will send control bits indicating for that no prefetching should be done. The FlexCache controller, when receiving this control pattern, will simply tell the cache controller to exit from the FlexCache state without executing any commands.

Custom Replacement Policy: Depending on the specific workload, certain replacement policies can improve performance. Custom replacement policies can be implemented by intercepting cache misses and using the FlexCache fabric to determine the block to be replaced. For example, the LRU (Least-Recently Used) replacement policy replaces the cache block that has been least recently used on a miss.

For the LRU replacement policy extension, the FlexCache fabric uses the SRAM blocks to maintain meta-data to remember the order of usage for cache blocks within each set. The event monitor is configured to send messages on all cache events so that the cache usage meta-data can be updated. The operation of the FlexCache controller is identical to the PLcache extension. On a miss, the FlexCache fabric first issues a `cinst_readMem` command followed by a `cinst_returnProc` with the unstall flag asserted. Meanwhile, the FlexCache fabric logic determines the least recently used cache block, which will be evicted, and sends this information to the FlexCache controller. The FlexCache controller finally issues a `cinst_writeCache` command using the set determined by the FlexCache fabric and returns control from the FlexCache state to the cache controller.

AES Accelerator: Since the FlexCache architecture uses a generic reconfigurable fabric, the FlexCache fabric can also perform arbitrary computations as long as they fit within the fabric. The FlexCache can be used as a memory-mapped computation accelerator. As an example, we used it to accelerate the Advanced Encryption Standard (AES) [4].

In order for a program to use FlexCache for acceleration, the program writes input data to the FlexCache fabric as a memory-mapped device. Since AES operates on 128-bit blocks of data at a time, our processor writes four 32-bit words to the FlexCache fabric. The processor then performs four 32-bit reads to get the result from the FlexCache fabric. These read operations wait until the computation is done so that the core does not need to continuously poll.

IV. Evaluation

This section evaluates the FlexCache architecture using silicon area, power consumption, and performance as metrics. In addition to the five extensions in Section III, the evaluation includes two more extensions, an MRU (Most-Recently Used) counter [2], which estimates the number of misses as a function of cache size, and an LIP replacement policy. The LIP policy is similar to the LRU replacement policy except when a new cache line is written, it is marked as the least recently used rather than the most recently used.

A. Methodology

For evaluation, we built an RTL prototype based on the Leon3 microprocessor [5]. Leon3 is a synthesizable VHDL model of a 32-bit in-order processor compliant with the SPARC V8 architecture. Leon3 includes L1 data and instruction caches without an L2 cache. We implemented the FlexCache design on a 32-KB 4-way set-associative L1 data cache with 32-B blocks. The cache uses a write-through no-allocate policy and a random replacement policy. In the FlexCache implementation, the command memory in the FlexCache controller has 1024 11-bit entries and the SRAM block in the FlexCache fabric is 1KB (1024-by-8).

To estimate the area, power consumption, and operating frequency of ASIC components such as the baseline Leon3 and the FlexCache additions, Synopsys Design Compiler (DC) was used with a 65nm IBM technology library. To estimate the same metrics for the FlexCache fabric portion of the extensions, we used Synplify Premier DP and Xilinx ISE, targeting the Xilinx Virtex-5 FPGA, which is also manufactured in a 65nm technology. The FPGA synthesis tools provide estimates for operating frequency and the number of LUTs required for each extension. The power consumption was calculated from the synthesis results using the Virtex-5 Power Spreadsheet. To compute a rough estimate of silicon area, we used an estimate of CLB tile area from the model by Kuon and Rose [6]. The model reports that the area of a CLB tile with 10 six-input LUTs in the 65nm technology node is approximately 8,069.46μm^2. We used this estimate and the total number of LUTs in our cache extensions to generate a CLB area estimate. The area of the command memory and SRAM blocks were generated from a commercial memory compiler and added to

TABLE II
AREA, POWER, AND FREQUENCY OF BASELINE LEON3, MODULES COMMON TO ALL FC EXTENSIONS, AND EACH EXTENSION SYNTHESIZED FOR ASIC AND FPGA. OVERHEADS ARE RELATIVE TO THE UNMODIFIED LEON3. IN PARENTHESES ARE THE TOTAL OVERHEADS INCLUDING COMMON MODULES.

Component	ASIC					FlexCache						
	Freq	Area		Power		Freq	Area Overhead			Power Overhead		
	MHz	μm^2	%	mW	%	MHz	μm^2	% (Total %)		mW	% (Total %)	
Baseline	465	817,320	-	373	-	-	-	-	-	-	-	-
FC Common	-	-	-	-	-	464	41,453	5.1	-	9	2.5	-
Event Counters	442	850,739	1.041	355	0.951	442	32,278	3.9	(9.0)	19	5.1	(7.5)
MRU Counters	446	853,519	1.044	371	0.995	250	81,502	10.0	(15.0)	23	6.2	(8.6)
PLcache	461	864,357	1.058	384	1.029	549	11,297	1.4	(6.5)	33	8.8	(11.3)
Prefetch	463	857,991	1.050	392	1.049	267	70,204	8.6	(13.7)	23	6.2	(8.6)
LRU	461	855,488	1.047	385	1.031	465	22,594	2.8	(7.8)	30	8.0	(10.5)
LIP	461	855,134	1.046	385	1.031	434	22,594	2.8	(7.8)	20	5.4	(7.8)
AES	450	858,495	1.050	369	0.990	173	289,694	35.4	(40.5)	26	7.0	(9.4)

obtain the total silicon area. Power estimates were obtained using a default toggle rate of 0.125 and the maximum FPGA clock frequency that is a power-of-2 ratio of the core clock based on the FPGA synthesis results.

To evaluate the impact of FlexCache extensions on overall performance, we performed RTL simulations of the entire system including processing core, caches, memory, and a FlexCache extension. The FlexCache fabric was run at a power-of-2 ratio of the core clock as was done for power analysis. The simulations ran a set of benchmark programs from MiBench [7] and other small kernels.

B. Area, Power, and Frequency

Table II shows the estimated area, power consumption, and operating frequency of the extensions synthesized for FPGA as well as for ASIC. The unmodified Leon3 with 32-KB L1 instruction cache and 32-KB L1 data cache can run up to 465 MHz and consumes 0.817 mm^2 of silicon area and 373 mW of power. The FlexCache modifications such as the FlexCache state in the cache controller, the event monitor, FlexCache controller, the FIFOs, and FPGA SRAM blocks which are common for all cache extensions, are implemented in ASIC and increase area by 0.041 mm^2 (5.1%) and power consumption by 9 mW (2.6%). The FlexCache fabric portions of the extensions were synthesized with the FPGA flow. The size of the FlexCache fabric that is required to support each extension ranges from 0.011 mm^2 to 0.290 mm^2 corresponding to 1.4% to 35.4% overheads over the unmodified Leon3. The additional power needed by the FlexCache fabric ranges from 19 mW to 33 mW which is 5.1% to 8.8% of the baseline. Thus, 0.290 mm^2 of FPGA fabric with a 33 mW power budget is enough to support any of the prototype extensions. Note that the compute intensive AES accelerator requires much more FPGA fabric than the other cache extensions. 0.082 mm^2 of FPGA fabric is enough to support the other extensions which corresponds to only 10.0% of the area of the unmodified Leon3. Combined with the fixed ASIC overheads, the FlexCache architecture requires 15.0% more silicon area than the baseline for these extensions.

While these overheads are noticeable relative to the Leon3 processor, we note that the absolute area and power consumption for the FlexCache architecture and extensions are quite small if compared to even simple commercial microprocessors. For example, each processing core in the UltraSPARC T2 occupies 12 mm^2 in the 65nm technology. MIPS R14000, which is fabricated in the 0.13μm technology, occupies 204 mm^2 and consumes 17W at 500 MHz. Overall, the results demonstrate that the FlexCache architecture can be implemented with small overheads in terms of the frequency, area, and power consumption.

For comparison, we also synthesized each cache extension with the Leon3 using the ASIC flow. The results, shown in Table II, show that the Leon3 with ASIC synthesized extensions require up to 5.8% more silicon area and consume up to 4.9% more power compared to the unmodified Leon3. The power estimates for some extensions are lower than that of the baseline because of the lower clock frequencies.

C. Performance Impact

There are three main sources of performance overhead in the FlexCache architecture: FIFO stalls, customizing cache operations, and background processing. If the FIFO from the cache controller to the FlexCache fabric is full, the controller stalls until an entry is free. However, in our experiments, the FIFO never filled up, even when the FlexCache fabric ran at half the core's clock frequency, because memory accesses do not occur every cycle. The customization of a cache operation can increase the number of cycles it takes. Background processing using the FlexCache controller will block the processor if it attempts to execute a memory instruction. These latter two overheads are discussed in more detail below.

Table III shows the cache hit and miss latencies for the implemented extensions. The number of additional cycles for background operations after a data value is returned to the core is shown in parentheses. During this period, the core can continue to run but will be stalled on a new memory access. As shown in the table, the event/MRU counters and the AES accelerator have no impact on the cache latencies because they never modify cache operations. PLcache, LRU and LIP increase the miss latency by 2 cycles. This increase comes from the fact that the current FlexCache design requires two separate commands for reading data from memory and returning to the processor. These extensions also require 2 more cycles of background operations to write the data

into the cache memory array, which is another FlexCache command. The prefetch extension does not change the cache latencies because both hits and misses are handled by the cache controller before initiating custom FlexCache operations. The background operations take up to 30 cycles and perform prefetching. While the majority of this background operation is the memory access latency, the remaining overhead is mostly due to communication from the cache to the FlexCache fabric. In extensions such as PLcache, LRU, and LIP, the FlexCache controller did not need information from the FlexCache fabric for the first command, a `cinst_readMem`, and so the communication latency could be hidden. However, for prefetching, the first `cinst_readMem` command needs information from the FlexCache fabric about whether to prefetch and if so, where to read data from. This difference can be seen in Fig. 4 and Fig. 5. As a result, there are several cycles of delay as messages are passed through the FIFOs to the FlexCache fabric and back.

Although FlexCache introduces additional delay in cache misses for some extensions, we found that the overall impact on performance is quite small because cache misses happen infrequently. Overall, PLcache only increases the execution time by up to 2.4% and on average 0.7%. These overheads would be lower for longer memory latencies. The Leon3 implementation used had a 16 cycle latency to main memory.

The prefetch and LRU/LIP replacement policies improve the miss-rates as shown in Table IV. Prefetching reduces the miss-rate by 2.3% on average, ranging from 10.1% reduction to 1.5% increase. For a synthetic benchmark with stride accesses, the prefetching extension reduces the miss-rate from 100% to 12.5%. LRU and LIP replacement policies reduce miss-rates by 1.6% and 1.0%, respectively, on average compared to the random replacement policy in the baseline. The miss-rate changes for LRU range from no change to 6.5% improvement. The changes for LIP range from 6.0% improvement to 0.7% degradation.

For the small benchmarks that we simulated, the miss-rate reduction was not always enough to offset the additional overheads. Prefetching showed execution time increases between 4.8% and 15.1% for the benchmarks in the table with an average of 8.4%. For the synthetic benchmark, performance was improved by 32.5%. The LRU and LIP extensions showed increased execution times from less than 0.1% to 1.9%. Both replacement policy extensions showed a 0.6% increase in execution time on average. We note that our benchmarks show very low miss-rates to begin with, leaving little room for improvement. In addition, the low main memory latency meant that there was a relatively small saving on a hit. As a result, the reduction in miss-rates was not enough to offset the overheads. However, the stride prefetcher and replacement policies have been shown to be effective for larger benchmarks and in fact are widely used. We believe that larger benchmarks will show overall performance improvements.

The AES accelerator extension showed a 78.4% improvement in execution time for the AES benchmark and no change for the performance of other benchmarks. Previous hardware AES accelerators have shown a range of performance improvements from 17% [8] to 97% [9] depending on the resources required and the baseline systems.

TABLE III
CACHE EVENT LATENCY IN CYCLES. IN PARENTHESES ARE THE NUMBER OF CYCLES OF BACKGROUND USE.

Extension	Hit Latency	Miss Latency
Baseline	1 (+0)	20 (+0)
Event Counters	1 (+0)	20 (+0)
MRU Counters	1 (+0)	20 (+0)
PLcache	1 (+0)	22 (+2)
Prefetch	1 (+30)	20 (+28)
LRU	1 (+0)	22 (+2)
LIP	1 (+0)	22 (+2)
AES	1 (+0)	20 (+0)

TABLE IV
MISSES PER 1000 MEMORY ACCESSES OF FLEXCACHE EXTENSIONS.

Benchmark	Baseline	Prefetch	LRU	LIP
aes	1.472	1.454	1.465	1.465
bitcount	1.803	1.694	1.803	1.803
basicmath	4.512	4.464	4.484	4.493
fft	4.032	4.094	3.768	3.789
gmac	4.020	4.082	3.924	4.047
stringsearch	414.337	372.397	410.668	410.668
sha	1.412	1.412	1.412	1.412
geomean	5.274	5.154	5.191	5.219

We note that the current FlexCache overheads can be reduced with further optimizations. For example, a single command to perform `cinst_readMem` and `cinst_returnProc` can eliminate the increase in the miss latency for the PLcache, LRU, and LIP extensions. The long background operations for prefetching can also be hidden if we augment the FlexCache implementation to operate in a non-blocking fashion using its own interface to memory. Prefetching could also be improved by providing a communication path that bypasses the FIFOs when they are empty, thus eliminating the need to first write to the FIFO and then read the message back out. These optimizations are possible to implement with more engineering effort and should cause minimal changes in the area, frequency, and power overheads. We leave the implementation of these optimizations as future work.

V. RELATED WORK

The integration of a reconfigurable fabric into a microprocessor has been extensively studied in the context of speeding up the main computation. Chimaera [10] and others [11], [12] use reconfigurable fabric as functional units. Garp [13] and OneChip [14] use a reconfigurable fabric as an on-chip accelerator. The Owl framework [15] uses reconfigurable fabric for hardware monitors. FlexCache also uses on-chip reconfigurable fabric which can be utilized for monitoring or as a computation accelerator. However, the main use of reconfigurable fabric in FlexCache is specifically to modify cache functionality. To the best of our knowledge, reconfigurable fabric has not been used previously in this context.

DeHon briefly mentioned using reconfigurable logic for monitoring and modifying cache events in his note [16].

However, the ideas have not been explored in detail.

Smart Memories [17] studied reconfiguring SRAM organizations and memory protocols through a programmable controller, which executes a set of primitive instructions to implement different protocols and memory operations. While the high-level goal of providing a flexible memory system is common, Smart Memories and FlexCache are significantly different in their target reconfigurability and approaches. For example, in Smart Memories, the programmable controller handles all accesses between the processors and memory elements, paying overheads on every memory access. Also, even transparent monitoring needs to be implemented by the controller in a sequential way. The FlexCache architecture supports parallel monitoring and reconfigurability of infrequent operations, and allows most accesses to be handled by the unmodified cache controller. FlexCache also provides a more general reconfigurable fabric that can also be used to accelerate computations.

Various architectures have been proposed to make caches more flexible. For example, reconfiguration of the number of ways in a set-associative cache [18], [19] and reconfiguration of the memory hierarchy [20] have been studied previously. These architectures can reconfigure the cache memory organization but do not allow new cache functionality to be added, which is the main goal of FlexCache. Ranganatahan et al. proposed cache partitions that can be configured for functions such as prefetching or scratch pad memory [21]. However, the approach requires the possible configurations to be decided at design time and fixed once fabricated. A previous project, also called FlexCache, moved parts of the cache system into software to create a more flexible cache [22]. However, the use of software can lead to higher overheads than our hardware-based approach.

VI. Conclusion

In this paper, we proposed a reconfigurable cache architecture named FlexCache. FlexCache is designed to support a broad range of cache extensions while maintaining the efficiency of normal cache operations. The architecture adds a reconfigurable fabric to the cache controller and enables transparent monitoring of cache operations and reconfiguration of infrequent events with low costs. By breaking the cache operation down into a set of basic commands, a reasonably-sized reconfigurable fabric can be used to extend cache functionality. The reconfiguration overhead is only visible when a particular event needs to be customized. The RTL prototypes show that the architecture can indeed support a range of cache extensions with minimal or no performance overheads and low impact on area and power.

Acknowledgments

This work was partially supported by the National Science Foundation grants CNS-0746913 and CNS-0708788, the Office of Naval Research under grant N00014-11-1-0110, the Army Research Office under grant W911NF-11-1-0082, and an equipment donation from Intel Corporation.

References

[1] D. Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, 1981.

[2] G. E. Suh, S. Devadas, and L. Rudolph, "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning," in *Proceedings of the 8th High Performance Computer Architecture*, 2002.

[3] Z. Wang and R. B. Lee, "New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[4] National Institute of Science and Technology, "FIPS PUB 197: Advanced Encryption Standard (AES)," November 2001.

[5] J. Gaisler, E. Catovic, M. Isomaki, K. Glembo, and S. Habinc, "GRLIB IP Core User's Manual," 2008.

[6] I. Kuon and J. Rose, "Area and Delay Trade-offs in the Circuit and Architecture Design of FPGAs," in *Proceedings of the 2008 ACM/SIGDA 16th International Symposium on Field Programmable Gate Arrays*, 2008.

[7] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *Annual IEEE International Workshop on Workload Characterization*, 2001.

[8] A. Irwansyah, V. P. Nambiar, and M. Khalil-Hani, "An AES Tightly Coupled Hardware Accelerator in an FPGA-based Embedded Processor Core," in *Proceedings of the International Conference on Computer Engineering and Technology*, 2009.

[9] A. Hodjat and I. Verbauwhede, "Interfacing a High Speed Crypto Accelerator to an Embedded CPU," in *Conference Record of the 38th Asilomar Conference on Signals, Systems and Computers*, 2004.

[10] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, "The Chimaera Reconfigurable Functional Unit," *IEEE Transactions on Very Large Scale Integration Systems*, 2004.

[11] R. Razdan and M. D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units," in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, 1994.

[12] H. Kim, A. Somani, and A. Tyagi, "A Reconfigurable Multifunction Computing Cache Architecture," in *IEEE Transactions on Very Large Scale Integration Systems*, 2001.

[13] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," in *Proceedings of the 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[14] R. D. Wittig and P. Chow, "OneChip: An FPGA Processor with Reconfigurable Logic," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[15] M. Schulz, B. S. White, S. A. McKee, H.-H. S. Lee, and J. Jeitner, "Owl: Next Generation System Monitoring," in *Proceedings of the 2nd Conference on Computing Frontiers*, 2005.

[16] A. DeHon, "Transit Note 118 Notes on Coupling Processors with Reconfigurable Logic," 1995. [Online]. Available: http://ic.ese.upenn.edu/transit/tn118/tn118.html

[17] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[18] C. Zhang, F. Vahid, and W. Najjar, "A Highly Configurable Cache Architecture for Embedded Systems," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[19] D. H. Albonesi, "Selective Cache Ways: On-Demand Cache Resource Allocation," in *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, 1999.

[20] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, 2000.

[21] P. Ranganathan, S. Adve, and N. P. Jouppi, "Reconfigurable Caches and Their Application to Media Processing," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[22] C. A. Moritz, M. Frank, and S. P. Amarasinghe, "FlexCache: A Framework for Flexible Compiler Generated Data Caching," in *Revised Papers from the Second International Workshop on Intelligent Memory Systems*, 2001.