

# CUSTOMIZABLE SECURITY-AWARE CACHE FOR FPGA-BASED SOFT PROCESSORS

Maciej Kurek, Ioannis Ilkos, Wayne Luk

Department of Computing, Imperial College London

## ABSTRACT

This paper describes a security-aware cache targeting *field-programmable gate array* (FPGA) technology. Our design is based on an architecture with a remapping table, which provides resilience against side-channel timing attacks. We show how this cache design can be optimised for FPGA resources by an index decoder with content addressable memory structure, which can be customized to meet various requirements. We show, for the first time, how our security-aware cache can be included in the Leon 3 processor, and its performance and resource usage are evaluated.

## 1. MOTIVATION AND CONTRIBUTIONS

The architectural schemes employed by modern processors, such as caches and branch predictors, are of great significance towards achieving speed and efficiency. However, their inherent characteristics also potentially expose security vulnerabilities. While these vulnerabilities vary in their impact and difficulty of exploitation, they can be as dangerous as the exposure of cryptographic keys. Data cache side-channel attacks are such an attack vector extensively studied with various mitigation techniques proposed. The problem with most of the methods of attack prevention is the speed and power efficiency degradation. A new way of countering the problem was proposed without the drawback of performance degradation [1]: a cache architecture that is not only security-aware, but also has a performance advantage over the traditional cache architecture. However, it has been designed using a transistor-level description, targeting implementation in *application-specific integrated circuit* (ASIC) technology. The main objective of this research was to design, present and evaluate a fast and resource-cheap method of countering the side-channel cache timing attacks targeting FPGA based devices.

This paper describes a customizable security-aware cache inspired by [1]. Our contributions are:

- An index-remapping circuit targeting FPGA technology, which is used in building customisable security-aware caches with different amounts of associativity (Section 3).
- Implementation of our design on the M5 simulator and Xilinx FPGAs, providing a security-aware cache

for the Leon-3 soft-processor with and without a *Memory Management Unit* (MMU) (Section 4).

- Evaluation of our approach, showing the resource usage and performance of the modified Leon-3 processor. In contrary to ASIC, the increased associativity not only leads up to 25% higher clock speeds, but also decreases the *lookup table* (LUT) and flip flop usage respectively by 7% and 30% at the possible expense of increasing miss rate, depending on specific applications (Section 5).

## 2. BACKGROUND

Side-channel cache attacks make use of timing information to infer additional information about an algorithm's state. There are three types of cache based attacks: trace-driven attack, the access-driven attacks and cache timing attacks.

Cache side-channel attacks exploit the variance in latency of memory accesses due to the cache. Knowing the cache architecture and the replacement policy of the target processor over a number of processor cycles, an attacker uses the channel created by the inherent cache properties in order to infer confidential data. There are many examples of various side-channel attacks, the most prominent being cache side-channel timing attack against AES [2, 3, 4].

Side-channel attacks pose a serious threat to modern computer systems. A range of potential countermeasures have been developed to counter the problem, but unfortunately their usability is severely limited due to performance degradation they impose on the system. For example, *Disabling the Cache* solves the problem of cache side-channel attacks but imposes large performance degradation. *Constant Timing programming* requires code recompilation and slows down the application, furthermore it only forces the attacker to use a larger number of samples to extract the desired information. *Cache Partitioning* prevents the attacker from information theft by denying the possibility to interfere with other users cache partition [5], but at the same time limits usable cache size.

The main novel idea behind the security-aware cache is the addition of a layer of indirection between the cache indices and the access of actual cache lines, so that each memory block can be assigned to an arbitrary cache line [1]. The

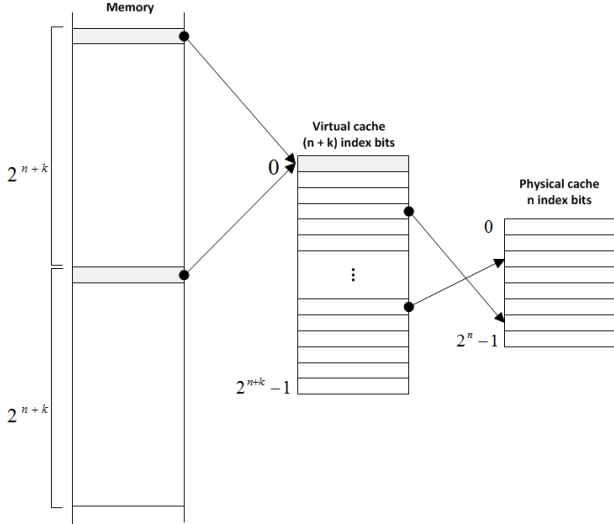


Fig. 1: Mapping memory to the physical cache.

index bits of the memory address are used to access a *remapping table* (RMT), which in turn returns the actual cache line number to be accessed. Additionally, this remapping stage allows us to have an arbitrarily large *virtual cache* accessed by the index bits of the address, which can be remapped to a smaller *physical cache*.

We utilize a physical cache indexed by  $n$  bits (having total size  $(2^n) \times \text{blocksize}$  bytes), but we extend the virtual cache by  $k$  bits. This provides greater resilience to conflict misses. The virtual cache does not exist. Instead, it is implemented by associating a *Line Number Register* (LNReg) to each physical cache line, which stores the  $(n+k)$  bits of the virtual cache line (the *index*) and is used at the remapping stage. The concept can be clearly seen in Fig. 1.

Along with the remapping data, the LNReg stores an additional field containing the context of the process the cache line belongs to. This is used in the *Security-Aware Random Replacement Algorithm* (SecRAND) [1] in order to protect processes from the external interference. There are three types of misses in SecRAND: (a) *Index misses* occur if there is no LNReg holding the needed index. (b) *Context misses* happen when there is an index match, but either the requested or stored line has a different context, then the context is protected. Protected block was written to cache by a process wanting to be safe against channel attacks from processes from a different context. (c) *An unprotected tag misses*. This means that the index is matched with a line and context but a tag does not match the requested address.

The replacement policies are different for each type of miss. In case of a regular miss, the behaviour of the cache is the same as a direct-mapped cache. In the case of a context miss, interference between different processes might leak information. Thus, we do not cache the incoming data from

the memory but send them directly to the CPU pipeline. However, in order to substitute for the cache miss that occurred, we evict a random cache line from our cache. This eviction also serves as a measure to avoid one process' monopoly on the cache. In case of an index miss we select a random cache line, evict it if necessary and update it.

The *security of the cache* stems from the fact that given an access to a cache line from a victim process, the attacker process (with a different context identifier) can observe the eviction of any cache line number with equal probability. The new cache provides security against all of the previously mentioned types of cache side-channel attacks. The *speed of the cache* is a result of the improved conflict miss ratio (due to the larger indices used to map the cache) and the implementation of the remapping stage with minimal overhead atop the address decoder, using transistor-level granularity.

### 3. SECURITY AWARE CACHE ON FPGA

Any device that is designed for ASIC can, in principle, be implemented on an FPGA device. The challenge is to make the resulting FPGA circuit efficient. The theoretical description of the security-aware cache address decoders implies a *content addressable memory* (CAM), which is used to find the correct index, and to issue index hit/miss. It is the analogue of the LN register array in the ASIC SecRAND [1] as well as the main cache sub-component differentiating the two designs.

The main challenges with security-aware cache FPGA implementation are related to the FPGA CAM implementation: high resource cost of CAM memories, multi-cycle read/write CAM operations, and increased critical path of CAM combinational word matching circuit.

To implement the LN registers, our design makes use of a CAM index decoder with a combinational word matching circuit. It leads to a cache design which is called the *single-set security-aware cache*. To counter the excessive resource demand of FPGA CAM implementation, we propose a new cache architecture which we call *L-associative security-aware cache*. In contrary to ASIC, the increased associativity not only leads to higher clock speed but also improves the resource utilization of the design.

#### 3.1. Single-set security-aware cache

In order to make the design portable, we make use of resources that are synthesizable on any FPGA platform. We design the CAM using a register file along with a combinational search circuit with an encoder (Fig. 2).

The CAM memory matching circuit extends cacheline addressing circuitry remapping the indices before they are used to access cache lines. The CAM matching is done in the same cycle as the access to cache line. *Note* that the CAM matching circuit effectively becomes part of the

cacheline addressing circuitry logic, therefore from control perspective read/write operations can be performed in the same way as in an unmodified cache. This means that any existing design can be extended with security-aware cache features without modifying the cache control.

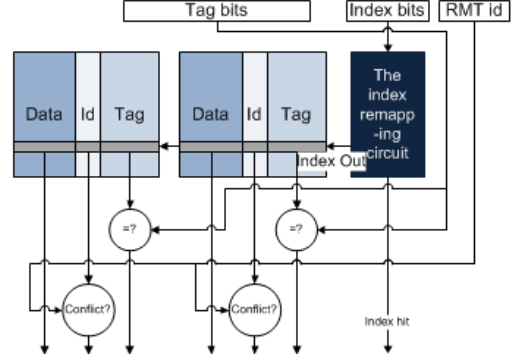
When *an access* is issued to the tag/data array, we do not use the usual address index of width  $n$  to access a tag/data line. We access the CAM memory decoder with index of width  $n+k$  where  $k$  is the extended index used to improve performance [1]. This index is later mapped to a tag/data array with  $n$  lines. When we issue a *write* to cache if the index that we search resides in the cache, we use the address provided from the CAM memory to access the appropriate cache line. If it does not reside in the CAM, we use the number generated from a random number generator to both index tag/data line as well as map the index to a line in the CAM memory. This way we ensure a single-cycle write to the tag/data lines and the CAM memory. To reduce the potential critical path, we register the write parameters, and perform write to CAM in the following cycle. Doing that we need to check whether a write is issued in the previous cycle, and whether there is an index match. The index from the write buffer has precedence over the CAM mapped indices as the written index is more recent.

### 3.2. Associative security-aware cache

The previously described cache can be expensive in terms of both LUTs and flip flops. The *L-Associative security-aware cache* (where  $L$  is the number of sets) is based on applying associativity principle to the single-set security-aware cache. As depicted in Fig. 3, we use one index remapping scheme to map indices in all of the sets. This decreases the resource usage up to  $L$  times compared to a single-set security-aware cache with the same capacity.

As the cache makes use of a number of sets, and only one line address decoder, the set validation mechanism is redesigned. The set validation control mechanism can be based on a number of different resources, and is highly dependent on the choice of the underlying CAM based decoder. We also extend the cache with a *valid table*. The valid table stores  $L$  bits per line, each indicating whether a set within a line is valid. In the ideal situation we can easily extend the tag array to incorporate valid tables, making a cheap and fast extension.

The drawback of this design is that when a cache line eviction occurs and an LNReg is updated, we invalidate all of the lines that made use of the former index. The hit rate degradation will depend on the application using the cache as well as on the number of sets; the higher the associativity, the higher the miss rate. The cache has the drawback of a larger block-size, without the advantage of spatial locality.



**Fig. 3:**  $L$ -associative security-aware cache index remapping circuit, with  $L=2$ . RMT denotes the remapping table.

### 3.3. Security Algorithm

The replacement policy depends on the decoder we use and on the cache write policy. If we use *write-through* policy, no dirty block has to be written back to memory during a protected line tag miss. *Write-back* policy is not recommended for this design. On a cache miss, all of the sets would have to be written back to memory. This could cause stalls, or require large buffers as the bus is optimized for small blocks.

There are two versions of the algorithm that we can use for the  $L$ -associative cache. The first is similar to SecRAND, and assigns all of the sets to one context. It is similar to increasing block size without increasing data pre-fetching. We implement this version of the algorithm.

In the second version each set can have a different context. On a tag miss involving a protected line, instead of writing to a randomly selected set we choose one that is unprotected (if there is one) and we write the new tag/data to that set. If all of the sets are used within one line, we follow the previous algorithm with the difference of invalidating one line within all of the sets. This version has performance advantage, although there could be of a possible side-channel achieved by over-accessing one line by an attacker. Furthermore this version is resource-wise more expensive as we have to store the context for every single set, as opposed to storing a line in the previous version.

## 4. IMPLEMENTATION

### 4.1. The modified M5 system

We build upon the M5 simulation suite. We evaluate the security-aware cache using both the Alpha and SPARC ISA. The latter would provide results for comparison with the FPGA implementation. We implement the new cache designs by modifying the cache subsystem. The security-aware replacement policy is implemented according to the cache tags interface mandated by the simulator. Additionally, the

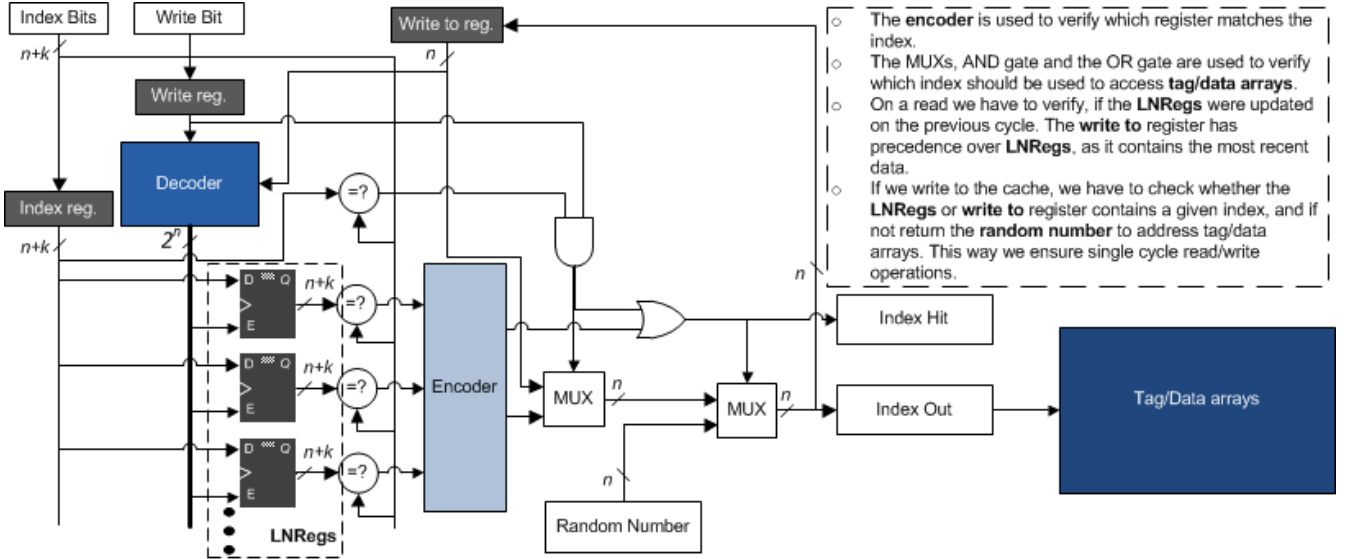


Fig. 2: Security-aware cache index remapping circuit. The  $E$  signal enables a register.

cache controller is enhanced to account for the extra information on the types of misses required, as well as the new write-back policies in case of an  $L$ -associative cache.

In order to perform the experiments, we use a series of benchmarks to model the system workload accurately: the Rijndael, Blowfish and Sha benchmarks of the MiBench suite [6] for embedded device benchmarking, the Qsort and FFT benchmarks of the MiBench suite, the Linpack linear equation solver and the Cache-chek benchmark.

The selection of the MiBench security suite is done on the grounds that we want to examine how the new cache designs fare specifically when it comes to cryptographic algorithms, while Qsort and FFT are selected based on the fact that they experience different behaviour during the cache operations. The rest of the MiBench suite are left out because they stress embedded CPU parameters, rather than the cache. Finally, the Linpack and cache-chek benchmarks are chosen because they are known to put a burden on the cache.

## 4.2. Security-aware cache implementation

The main elements of our security-aware cache are the *index remapping circuit*, *validation mechanism*, and *replacement algorithm*.

While there is an elegant way of performing remapping on top of the address decoder circuits in ASIC [1], FPGAs do not support transistor-level granularity. Since FPGA *block random access memory* (BRAM) already contains its own decoding logic (which cannot be bypassed), we remap an index address to another index address to be decoded by the BRAM circuits, instead of remapping the index address directly to a cache line selector signal.

In order to create a portable design, we implement the

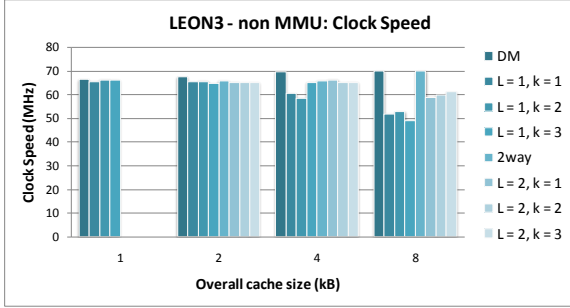
remapping stage using flip-flops. Although *intellectual property* (IP) CAM memories are available for nearly all FPGA platforms, they are based on edge-triggered circuits [7] which can lead to additional cycle penalty. This renders the cache unportable, as it necessitates a redesign of the cache controller while the latency of all memory instructions is increased by one or more cycles, degrading the effective *Instructions Per Cycle* (IPC) metric.

Given that we use memory-based remapping, we must have a way of ensuring the validity of the remapped data for all set blocks. In order to store validity bits for each of the blocks for a cache set, we have two options: storing them with the LNRregs or using the cache tag's validation table. The first solution is more expensive since it can greatly increase the number of required flip flops of the design.

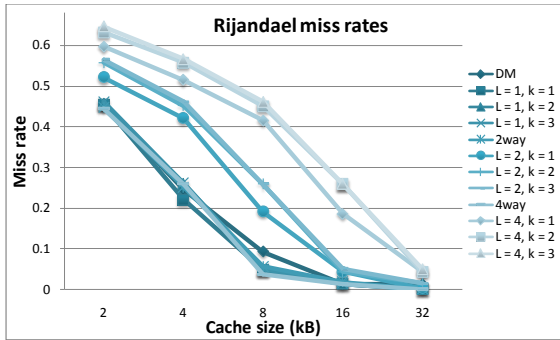
The next section describes the cache controller of the Leon 3 soft processor to adopt our security-aware cache for both MMU and non-MMU enabled designs, to disambiguate between regular and index misses. Additionally, we shall extend the MMU cache controller to identify context misses.

## 4.3. Leon 3 processor

LEON3 is an open source, SPARC v8 compliant CPU core written in VHDL. It is designed primarily for embedded applications, combining high performance and low power consumption. The processor uses a Harvard architecture, with separate configurable data and instruction caches. The data cache uses a write-through policy with no-allocate on write. LEON3 either works without an MMU, or with the reference SPARC v8 MMU [8]. Therefore, its IP contains two different cache controllers, with or without MMU support, interfacing with the cache memory.



**Fig. 4:** Miss rates for 16kB cache across all benchmarks. DM, 2way, 4way caches are not security-aware.



**Fig. 5:** Rijandael miss rates for different cache parameters.

We modify the LEON3 cache memory and data cache controller to incorporate our changes. Given that the full security algorithm requires context information, we focus our design on the single core MMU-enabled LEON3 processor and use the single core non-MMU version as a testbed platform for the remapping stage alone (i.e. the cache coherency algorithms have not been modified).

The remapping stage is inserted as a layer between the cache index generation of the cache controller and the actual BRAM memories holding the tags / data of the cache. Subsequently, we modify the data cache controller in order to implement the new cache replacement algorithm. We enhance the cache controller control register with two extra fields, indicating if we are returning from an index miss or a context miss. In order to incorporate the remapping validation mechanism in the tag array, we use the pre-existing block valid bits (to avoid further costs) and modify them accordingly: whenever we require a set invalidation, we reset the valid bits of all related cache blocks.

## 5. EVALUATION

### 5.1. Simulation

We present how the customization of the security-aware cache using  $L$  (amount of associativity) and  $k$  (number of bits by

which we extend the index) as well as cache size affects the cache hit-rate. We observe that the single-set security-aware cache does not have a negative impact on cache performance. Actually it reduces the miss rate due to the reduction of context misses, achieving the optimal miss rates for the cache sizes of 16 kB and 32 kB.

We observe that increasing the associativity of the security-aware cache tends to degrade cache performance. This is expected due to the eviction of multiple blocks in case of index misses. We also observe, for Rijandael, that increasing the  $k$  parameter does not improve the hit rate; in most cases it actually degrades it. One reason is that the random set to be evicted for each cache miss might be needed again and if so, the eviction will result in additional cache misses. Using the  $L$ -associative cache (which evicts multiple blocks) increases the amount of data evicted, thus the probability of such additional misses, resulting in larger performance penalty of using higher  $k$  values.

The cache miss rates across all benchmarks for different 16 kB cache designs are depicted in Fig. 4. In some cases, our security-aware caches do not degrade the performance to a noticeable extent, while in some cases (such as FFT) they are more effective. The Rijndael, Qsort and Linpack benchmarks suffer noticeably when we use the  $L$ -associative cache due to index miss evictions of multiple cache blocks.

### 5.2. Extended Leon 3 performance and customization

For FPGA synthesis, placement and routing of the Leon 3 processor we use Xilinx ISE 11.5. The testbed FPGA platform is based on Xilinx Virtex 4 XC4VFX12. The selection of this FPGA platform since it is a low-resource FPGA that can stress our designs and allow us to pinpoint their disadvantages. The user can customize the cache with  $k$  and  $L$  parameters in addition to the original Leon 3 parameters.

As depicted in Fig. 8, we observe a significant linear increase of the LUT usage as the cache size increases. The major causes of this increase are the additional logic required for the encoding and decoding of the cache indices, as well as the additional comparators required to compare the larger cache indices. This problem is alleviated by using an  $L$ -associative cache because for a given overall cache size, the number of cachelines is reduced.

We compare  $L=(1,2,4)$ , 8 kB,  $k=1$  caches. The 4-set cache achieves a 25% improvement and the 2-set achieves a 11% improvement in terms of clock speed over the 1-set cache. The flip flop utilization also improves (4068, 3060, 2879), and the 2-set cache achieves the best LUT utilization (7311, 6728, 6846). We notice a speed advantage of the 4 kB security-aware cache over the 2-set ordinary cache in terms of clock speed for  $k=1$  and 3. The clock speed of the 8 kB security-aware 4-set cache is just 6% slower than the corresponding ordinary cache.

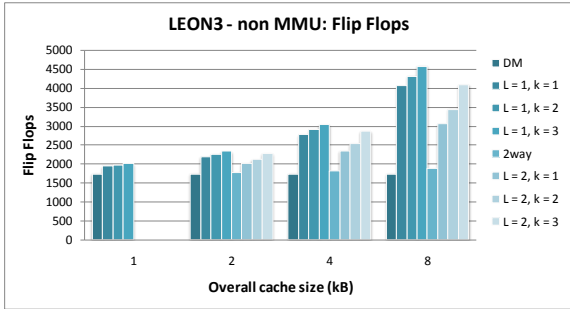


Fig. 6: Flip flop usage, without MMU

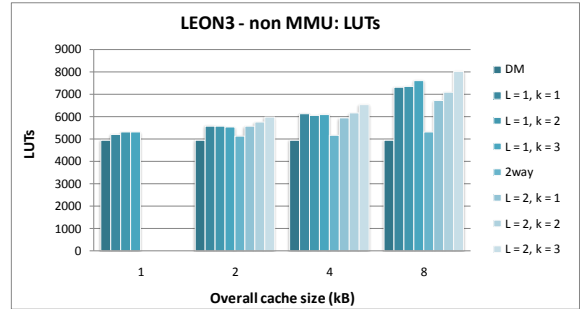


Fig. 8: LUT usage, without MMU

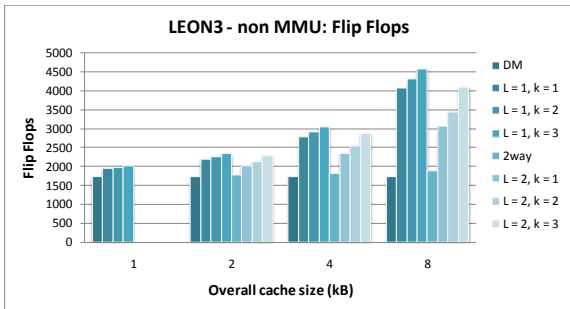


Fig. 7: Clock Speed, without MMU

We compare the two possible cache validation table implementations, BRAM and flip flop. BRAM implementation has an advantage in terms of achievable clock speed and resource usage. For an 8kB cache with  $L=2$  and  $k=3$ , 8 kB cache the difference between BRAM and flip flop based design in terms of LUTs, flip flops and clock speed is (8691, 4351, 59 MHz) and (8028, 4092, 61 MHz), at the slight expense of BRAMS.

Due to the additional index bits required for the remapping stage, flip flop usage increases with  $k$ . The value of  $k$  should be either 1 or 2, as increasing it beyond that imposes resource cost with no hit rate improvement (Fig. 4-8).

Although the resource cost of the MMU and non-MMU version of the design is different, they follow similar patterns in terms of resource usage when adjusting the  $k$  and  $L$  parameters.

## 6. SUMMARY

This paper describes single-set and associate security-aware caches targeting FPGA technology. We show, for the first time, how such caches can benefit the Leon 3 soft processor. The security-aware cache has several parameters and features which result in a large design space. We know that the impact of  $k$  and  $L$  on the design performance is non trivial and application dependent, but there are a few generalizations which we can make. For example, reducing the  $L$  parameter will reduce the amount of resources used to im-

plement remapping stage linearly, while increasing the  $k$  parameter will increase the amount of decoding logic and storage linearly. Increasing the block size will reduce resource required, since fewer remapping registers will be needed. Further work includes evaluating the resource and performance impact of various combinations of the cache parameters with a wide range of benchmarks, and exploring techniques for optimising such parameters automatically to meet application-specific requirements.

**Acknowledgement.** The support of UK EPSRC, Xilinx, and the HiPEAC Network of Excellence is gratefully acknowledged.

## 7. REFERENCES

- [1] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *MICRO 41*. IEEE Computer Society Press, 2008, pp. 83–93.
- [2] D. Bernstein, "Cache-timing attacks on AES," <http://cr.yp.to/antiforgery/cachetiming-20050414.pdf>.
- [3] J. Bonneau and I. Mironov, "Cache-Collision Timing Attacks against AES," in *Cryptographic Hardware and Embedded Systems – CHES 2006*, pp. 201–215.
- [4] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," in *Topics in Cryptology - CT-RSA 2006, The Cryptographers Track at the RSA Conference 2006*. Springer-Verlag, pp. 1–20.
- [5] D. Page, "Partitioned cache architecture as a side-channel defense mechanism," <http://eprint.iacr.org/2005/280.pdf>, 2005, retrieved June 15, 2010.
- [6] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," *IEEE International Workshop on Workload Characterization, WWC-4*, pp. 3–14, 2001.
- [7] J.-L. Brelet, "Xilinx Application Note XAPP201: An Overview of Multiple CAM Designs in Virtex Family Devices," pp. 1–6, Dec 1999.
- [8] D. Weaver and T. Germond (eds.), *The SPARC Architecture Manual*, Prentice Hall, 1992.