FlexSig: Implementing Flexible Hardware Signatures

LOIS OROSA, ELISARDO ANTELO, and JAVIER D. BRUGUERA,
University of Santiago de Compostela

With the advent of chip multiprocessors, new techniques have been developed to make parallel programing easier and more reliable. New parallel programing paradigms and new methods of making the execution of programs more efficient and more reliable have been developed. Usually, these improvements require hardware support to avoid a system slowdown.

Signatures based on Bloom filters are widely used as hardware support for parallel programing in chip multiprocessors. Signatures are used in Transactional Memory, thread-level speculation, parallel debugging, deterministic replay and other tools and applications. The main limitation of hardware signatures is the lack of flexibility: if signatures are designed with a given configuration, tailored to the requirements of a specific tool or application, it is likely that they do not fit well for other different requirements.

In this paper a new hardware signature organization, called *Flexible Signatures* (*FlexSig*), is proposed. *FlexSig* can change dynamically the resources assigned to a given signature and the number of signatures in the system, by redistributing the available hardware resources according to the system requirements. This allows higher flexibility than with traditional fixed-resources signatures based on Bloom filters, while maintaining a low false positive rate.

FlexSig has been evaluated by comparing it with signatures based on parallel Bloom filters, and we conclude that *FlexSig* outperforms (in terms of false positive rate) conventional parallel Bloom filters in most cases, due to its ability to use all the signature resources available.

Categories and Subject Descriptors: B.3.0 [**Memory Structures**]: General; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors); C.1.4 [**Processor Architectures**]: Parallel Architectures

General Terms: Design, Algorithms, Performance, Experimentation

Additional Key Words and Phrases: Signatures, Bloom filter, scalability, flexibility, CMPs, multithreading, transactional memory

ACM Reference Format:
Orosa, L., Antelo, E., and Bruguera, J. D. 2012. FlexSig: Implementing flexible hardware signatures. ACM Trans. Architec. Code Optim. 8, 4, Article 30 (January 2012), 20 pages.
DOI = 10.1145/2086696.2086709 http://doi.acm.org/10.1145/2086696.2086709

1. INTRODUCTION

New parallel programming techniques are needed to take full advantage of emergent multicore architectures with several cores and shared memory. Unlike sequential

This work was supported in part by the Ministry of Education and Science of Spain and FEDER funds under contract TIN 2010-27541 and by Xunta de Galicia under contract 2010/28.
J. D. Bruguera is also with the "Centro de Investigación en Tecnologías de la Información (CITIUS)", University of Santiago de Compostela, Spain.
E. Antelo and J. D. Bruguera are members of the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC).
Author's addresses: L. Orosa, E. Antelo, and J. D. Bruguera, Department of Electronics and Computer Science, University of Santiago de Compostela, Spain; email: {lois.orosa, elisardo.antelo, jd.bruguera}@usc.es.

programming, parallel programming may introduce unexpected behaviors due to the concurrent execution of threads and due to the difficulties in understanding the low level behavior of the architecture by programmers. Thus, programming becomes more complicated in this environment and prone to introduce errors (data races, atomicity violations, deadlocks, etc). In addition, these errors are usually non deterministic and, therefore, are hard to locate, reproduce and fix. Consequently, the research community is making a great effort to allow an easier transition to parallel programming through the development of new programming paradigms, tools and hardware support.

Traditional low level synchronization methods based on locks can be used, but some mechanisms for detecting and tolerating synchronization bugs have to be introduced [Adve et al. 1991; Choi et al. 2002; Netzer and Miller 1989; Netzer and Miller 1991; O'Callahan and Choi 2003; Prvulovic and Torrellas 2003; Yu et al. 2005; Zhou et al. 2007; Lucia et al. 2008; Ratanaworabhan et al. 2009; Muzahid et al. 2009; Tuck et al. 2008]. Alternatively, programming abstractions for shared memory multiprocessors, such as Transactional Memory [Herlihy and Moss 1993; Ananian et al. 2005; Hammond et al. 2004; Yen et al. 2007] or Speculative Multithreading [Ceze et al. 2006], have been proposed to make the programming of multicore processors easier and more reliable. However, despite the fact that these new parallel programing abstractions improve traditional synchronization methods, they are not error free, and techniques for detecting and tolerating concurrency bugs have to be used as well [Lev and Moir 2006].

These methods often need hardware support to improve performance. A widely used hardware resource are signatures [Bloom 1970; Ceze et al. 2006; Ceze et al. 2007]. Signatures are a low-cost hardware resource to keep an unbounded number of memory addresses in a fixed register space. When an address is checked for ownership, the signature can report a false positive, but it must never report a false negative. Obviously, the larger the number of addresses inserted in the signature, the larger the false positive rate.

Signatures are also popular in other domains, for example in network processors. Packet classification [Chang et al. 2004] uses signatures to improve performance by relaxing accuracy. In keyword searching [Reynolds and Vahdat 2003] signatures are used to determine remote set intersections. They are also used for locating and routing in peer-to-peer location mechanisms [Rhea and Kubiatowicz 2002]. In web cache sharing protocols [Fan et al. 2000] signatures host a summary of the URLs cached documents.

To meet the expectations of future multicore architectures, several hardware signatures might be necessary. Thus, in a general purpose microprocessor, signatures might be required in several applications, each one with very different configuration requirements, in terms of the number of signatures and its size.

Conventional fixed-size and fixed-number signatures are very inflexible and hard to adapt to different scenarios. For instance, conventional Transactional Memory implementations need two signatures per thread, but other applications or tools could require a different number of signatures. In addition, the size of the signature must be chosen carefully. The depends both on the signature size and on the number of addresses inserted; consequently, it depends on the application. A large size signature would probably be enough for most of the applications, but it consumes significant hardware resources. Alternatively, a small size signature needs much fewer hardware resources, but the false positive rate could be too high for the most demanding applications. Therefore, traditional signatures overestimate the number and size of the signatures to deal with those different requirements.

In this paper we propose a new hardware signature organization that we call *Flexible Signature (FlexSig)*, that can dynamically change the number and size of the signatures. Hence, *FlexSig* can efficiently manage situations where the number of required signatures is unknown a priori or variable.

The target of *FlexSig* is to obtain a more flexible signature system than traditional signatures based on Bloom filters, by using all the hardware resources independently of the number of threads running in the system. By using a fixed signature space, *FlexSig* is able to manage a high number of requests simultaneously when signatures are highly demanded, and achieve a low rate of false positives when there are few concurrent requests in the system.[1] Additionally, it provides fault tolerance because if some signature registers fail, *FlexSig* can continue operating just by invalidating the faulty registers and redistributing the remaining registers among the threads.

For instance, in a processor with 16 cores and 1 thread per core demanding from 0 to 4 signatures each, with at least one thread requiring at least one signature, the maximum number of signatures required simultaneously is 64 (4 signatures $\times$ 16 threads), and the minimum is 1, (only 1 thread running and it demands just 1 signature). Let us assume that the most usual situation is to need 8 signatures simultaneously; for example, 4 threads with 2 signatures each. Using signatures based on Bloom filters, 64 signatures of a given size (to be determined to minimize the false positives rate) are required to properly manage the worst case. However, with *FlexSig* it is possible to focus on the most common situation: a register space equivalent to 8 Bloom filters, but with enough flexibility to keep up to 64 concurrent signatures. Of course, the larger the number of signatures, the lower the number of registers assigned to each one. With this configuration, *FlexSig* behaves efficiently for the most common case of 8 simultaneous signatures. When only 1 signature is required, *FlexSig* provides a large signature with a size 8 times larger than a regular signature. In the rare case of requiring 64 signatures, *FlexSig* provides it anyway, but with a smaller size and, therefore, a higher false positives rate.

FlexSig is evaluated by comparing with fixed signatures based on Bloom filters in a Transactional Memory (TM) System. The results show that *FlexSig* outperforms the fixed signatures, with significant reduction in the false positive rate, specially when the number of threads does not match the maximum hardware threads.

There are other papers focused on improving signatures [Yen 2009; Shenghua et al. 2009; Yen et al. 2008; Almeida et al. 2007] and even to propose a scalable scheme [Almeida et al. 2007] but, to the best of our knowledge, this is the first work focusing on improving the flexibility of hardware signatures.

The rest of the paper is organized as follows. In Section 2, hardware signatures are reviewed. In Sections 3 and 4, our approach to flexible signatures is presented and some implementation issues are discussed. In Section 5, *FlexSig* is evaluated. Finally, in Section 6 related work is discussed and the conclusions are summarized in Section 7.

2. BACKGROUND ON HARDWARE SIGNATURES

In this section we focus on hardware signatures, although signatures can be implemented in software or hardware. Signatures are used to keep the set of addresses generated by the different cores or threads in the system and to detect conflicts in the memory accesses among cores/threads. Hardware signatures are composed of a *m-bit* register and several hash functions where the addressed are hash-encoded and stored.

The basic operations which must be supported by a signature are: *insert* a new address, *check* if an address is already in the signature (*conflict*) and *clear*. Due to aliasing, a conflict may be detected in the check operation when no actual conflict exists (false positive), but no conflicts are missed, that is, false negatives are not possible. There are different implementations of signatures, but here we focus on signatures

[1]Note that we talk about concurrent signatures instead of concurrent threads, since threads do not require signatures all the time. Therefore, the number of concurrent signatures are usually smaller than the number of concurrent threads.

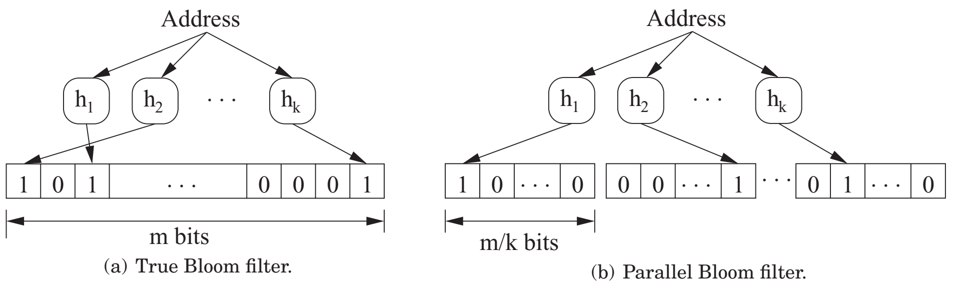


Fig. 1. Types of Bloom filters.

based on Bloom filters [Bloom 1970] because they are widely used and are the base of *FlexSig*.

2.1. True Bloom Filters

Figure 1(a) shows the basic scheme of a signature based on Bloom filters [Bloom 1970]. It is composed of a *m-bit* register and one or more hash functions $[h_1 \cdots h_k]$. This kind of signature, which is based on a single Bloom filter, is called *True Bloom signature*. In an insert operation the hash function receives an address and sets a bit in the register. To check if an address is stored in the signature, the address is hashed and compared with the content of the register. To clear the signature all bits of the register are set to zero.

The most critical design decisions in a true Bloom filter are the size of the register (m) and the number of hash functions (k). Large registers decrease the probability of a false positive, but increase the hardware resources and power required. On the other hand, the probability of false positives depends also on the number of hash functions and the number of elements inserted [Sanchez et al. 2007; Bloom 1970].

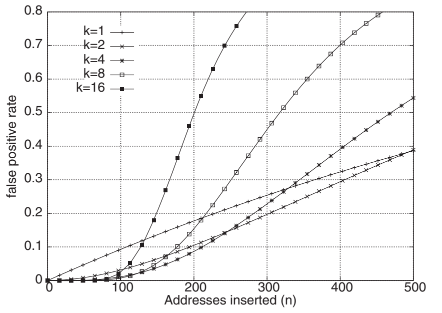
The number of false positives is influenced as well by how the hash functions are implemented. Very simple implementations are not efficient in terms of false positives. A very popular and widely used family of hash functions is H_3 [Carter and Wegman 1977; Ramakrishna et al. 1997]. H_3 requires additional hardware, specifically, for n-bit addresses a tree of $n/2$ two-input XOR gates for each bit of the hash function is needed, but it achieves a better false positive rate thanks to uncorrelated and uniformly distributed hash values.

2.2. Parallel Bloom filters

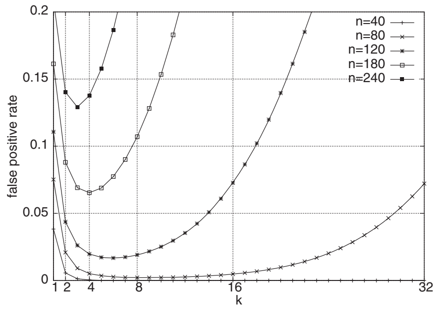
Figure 1(b) shows an improvement of the the true Bloom filters, the parallel Bloom filters [Sanchez et al. 2007; Chang et al. 2004]. In this case, the m-bit register is split into k m/k-bit registers, each with a hash function. In this way, each hash function operates on different parts of the m-bit register. Hence, the Parallel Bloom filter can be seen as k true Bloom filters with one hash function and a m/k-bit register. The insert operation hashes the address and inserts one bit in each m/k-bit register. The advantage of parallel signatures is that hash functions are simpler and are implemented with fewer resources. Also, each of the individual Bloom filters can be single-ported, thereby greatly reducing the hardware area/power/latency.

The false positive rate depends on the size of the signature m, the number of hash functions k and the number of inserted addresses n. and is given by the following expression [Sanchez et al. 2007; Chang et al. 2004]:

$$P = (1 - (1 - k/m)^n)^k$$



(a) False positive rate for parallel Bloom filter with $m = 1024$, $k = \{1, 2, 4, 8, 16\}$ and n in the x axis.



(b) False positive rate for parallel Bloom filter with $m = 1024$, $n = \{40, 80, 120, 180, 240\}$ and k in the x axis.

Fig. 2. False positive rate for parallel Bloom filters.

Figures 2(a) and Figure 2(b) show the false positive rate for $m = 1024$ and different values of k and n. It can be seen that a large k degrades the false positive rate quickly as the number on inserted addresses increases. The optimum value for k is between 3 and 8.

3. FLEXIBLE SIGNATURES

This section describes the *FlexSig* scheme. The overall organization and the strategy for allocating signatures are described in detail and an implementation is outlined. Moreover, the software interface and the actions to be taken in case of overflow and hardware failure are discussed.

3.1. Overview

FlexSig is based on parallel Bloom filters (Section 2), but introducing mechanisms to use all the resources in the signatures as much as possible and with a large flexibility to adapt to different signature demands, allowing a better efficiency and reconfigurability.

Figure 3 shows the block diagram of *FlexSig*. It is composed of T Bloom filters, each one composed of a M/T-bit register (M is the total size of the *FlexSig*) and a hash function, that can host between 1 and T signatures, each signature being composed of one or more Bloom filters. Each Bloom Filter has an identifier (ID) of the signature to which it belongs. The number of Bloom filters assigned to each signature depends on the number of signatures allocated. Moreover, the resources assigned to a given signature may change dynamically with time. $h_1, h_2, \ldots, h_T$ are independent H_3 hash functions, each operating on one register. The registers in *FlexSig* are usually relatively small (for instance, 64 bits), because a signature is to be composed of several of them. Every time *FlexSig* receives a request to insert a new address in one of its signatures, each hash function assigned to the signature sets one bit in its register. On the other hand, to check if an address is already in the signature, all the bits read by the corresponding hash functions should be 1. Deallocation requests clear all the IDs and registers assigned to the signature.

Each time a new signature allocation request arrives, *FlexSig* assigns k Bloom filters, $k \leq T$, to the new signature. Then, the k Bloom filters operate as a parallel Bloom filter inside *FlexSig*. The number of Bloom filters assigned depends on the current resource availability, that is, on the already allocated Bloom filters to previous signatures. If the hardware resources in *FlexSig* are fully used by previous signatures, *FlexSig* have to free several Bloom filters, already assigned, to allocate the new signature. This means that *FlexSig* has to reduce dynamically the size of any signature by releasing Bloom

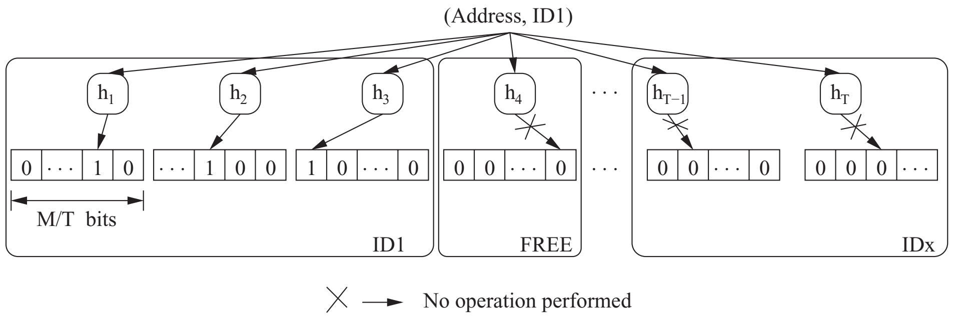


Fig. 3. Block diagram of *FlexSig*. Each allocated signature in *FlexSig* has a variable number of hash functions k between 1 and T, depending on the number of signatures allocated concurrently. Each signature is identified with an ID. The total register space assigned to each signature is $m = k * M/T$ bits.

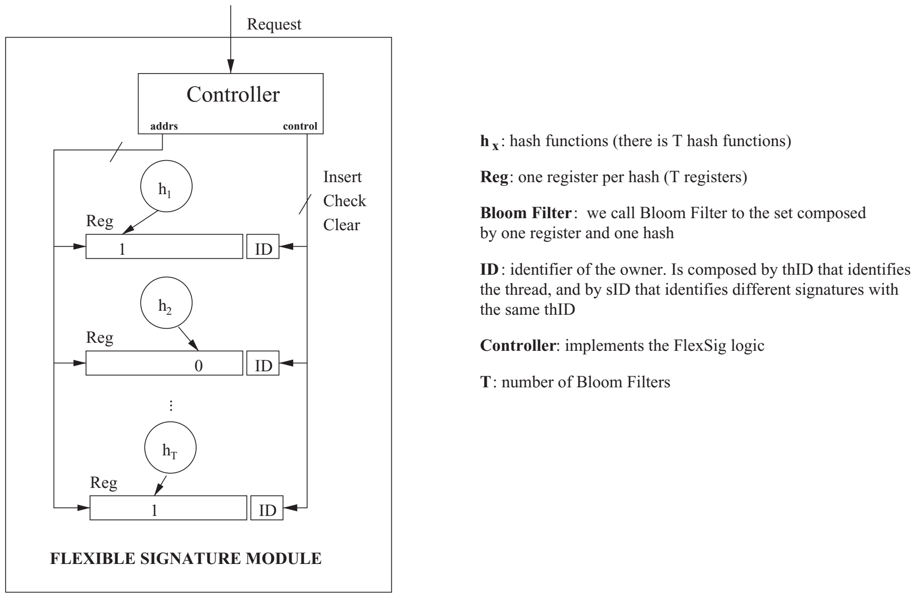

Request
Controller
addrs
control
Insert
Check
Clear
h_1
Reg
1
ID
h_2
Reg
0
ID
h_T
Reg
1
ID
FLEXIBLE SIGNATURE MODULE
h_x: hash functions (there is T hash functions)
Reg: one register per hash (T registers)
Bloom Filter: we call Bloom Filter to the set composed by one register and one hash
ID: identifier of the owner. Is composed by thID that identifies the thread, and by sID that identifies different signatures with the same thID
Controller: implements the FlexSig logic
T: number of Bloom Filters

Fig. 4. *FlexSig* module architecture.

filters. In this case, the false positive rate may increase, but false negatives are never produced.

Figure 4 shows the *FlexSig* module architecture. As said before, there are T Bloom filters, each one composed of a register and a hash function. Attached to each register there is a thread identifier (thID) and a signature identifier (sID), used to identify the registers assigned to a given signature. Therefore, thID and sID are $log_2(num\_threads)$-bit and $log_2(max\_sigs\_per\_thread)$-bit wide, respectively. Both thID and sID form the signature owner identifier (ID)

The minimum number of Bloom filters per signature in *FlexSig* is $T/num\_max\_concurrent\_sigs$ (the Bloom filters are distributed equally among signatures), and the maximum size is T (when only one signature is allocated).

The controller implements the allocation algorithm and the rest of functions needed for the correct operation of *FlexSig*. The complexity and efficiency of the controller depends on the algorithm to allocate and to free signature registers.

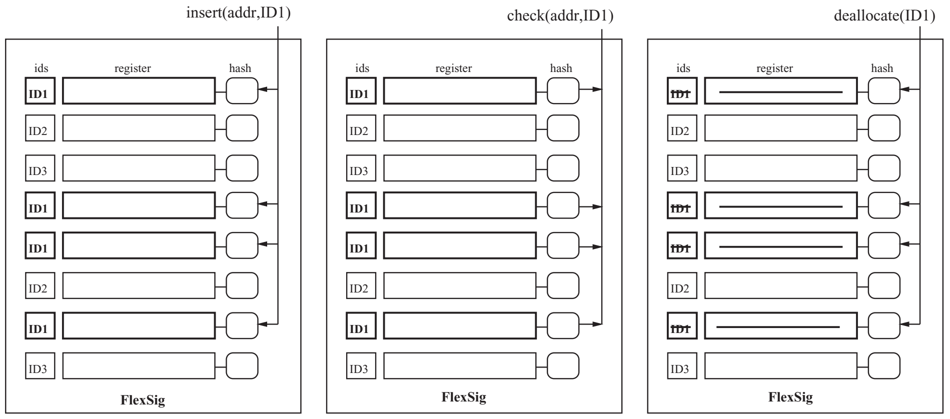


Fig. 5. Insertion, check and deallocation request in *FlexSig*.

Figure 5 shows how to perform the insertion, check and deallocation requests. The insertion request must include the ID for the signature, so that the address is only inserted in the registers that matches this ID. The check operation is very similar to the insertion operation, but it is read-only. The deallocation consists of clearing the registers and IDs.

3.2. Allocation Algorithm

The allocation algorithm is required to make room for a new signature and to free the space occupied by a signature once it is not needed anymore. This algorithm may be very complicated, for example, by defining priorities to assign more or less Bloom filters depending on the requirements of the allocated signature. In this work we show a simple allocation algorithm with no priorities. The algorithm with priorities requires a deep study of the design space (number of priorities, resources assignment depending on the priority, etc) to have an efficient implementation, and it will be developed in a future work.

To perform the description of the allocation algorithm, the following parameters are defined.

—*n_sig*: Number of signatures allocated in *FlexSig*.
—*n_reg_free*: number of free Bloom filters in *FlexSig*.
—*n_to_free*: number of Bloom filters to be freed by the allocation algorithm.

Three different situations are possible when a thread tries to reserve space for a new signature in *FlexSig*: (1) *FlexSig* is empty, (2) *FlexSig* is full, or (3) *FlexSig* is partially full.

(1) *FlexSig* is empty ($n\_sig = 0$, $n\_reg\_free = T$). All the resources of *FlexSig* are assigned to the new signature. This is one of the basic principles of *FlexSig*: if there are free resources, take as much as possible.
(2) *FlexSig* is full ($n\_sig = T$, $n\_reg\_free = 0$). The controller must free space in *FlexSig* when it is necessary to allocate a new signature. Then, the other signatures in *FlexSig* are made smaller by reducing the number of Bloom filters per signature. The release of one or several Bloom filters assigned to a given signature may increase the false positive rate but false negatives are never produced, because all the hash functions are independent and all the registers in a signature have the information corresponding to every address inserted.

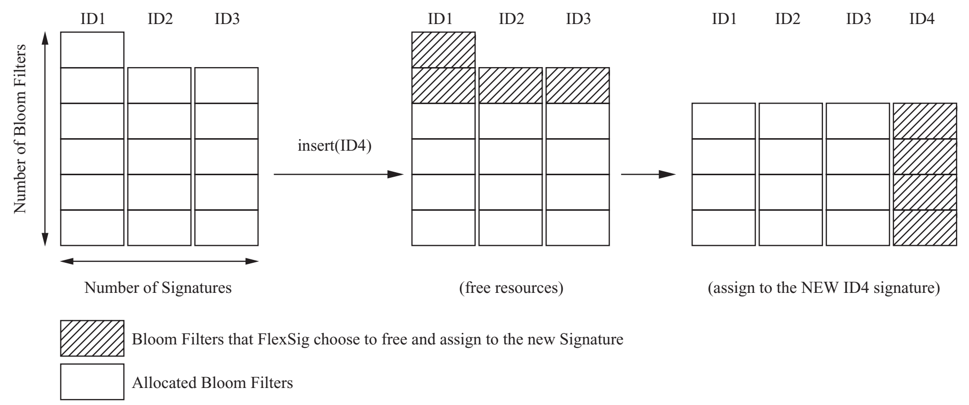

ID1
ID2
ID3
ID4
Number of Bloom Filters
Number of Signatures
insert(ID4)
(free resources)
(assign to the NEW ID4 signature)
Bloom Filters that FlexSig choose to free and assign to the new Signature
Allocated Bloom Filters

Fig. 6. Example of the allocation algorithm when *FlexSig* is full (example for $T = 16$).

The number of Bloom filters to free is given by

$$n\_to\_free = \left\lfloor \frac{T}{(n\_sig + 1)} \right\rfloor \quad (1)$$

The free filters are assigned to the new signature. Therefore, the filters are redistributed equally among all the signatures including the new one.
Figure 6 illustrates an example of the allocation algorithm in a *FlexSig* module composed of 16 Bloom filters. Initially, there are 3 signatures allocated, one of them has 6 filters assigned and the other has 5 filters. When a new signature is allocated, $n\_to\_free = 4$, and the controller clears filters and tries to assign the same number of filters to every signature.

(3) *FlexSig* is partially full. The controller must decide whether the free resources are enough to allocate a new signature or if additional resources are needed. In the latter case, some filters should be freed and assigned to the new signature. If $n\_reg\_free < n\_to\_free$, the controller frees $(n\_to\_free - n\_reg\_free)$ Bloom filters as explained for the case when *FlexSig* is full. On the other hand if $n\_reg\_free \geq n\_to\_free$ all the available Bloom filters are assigned to the new signature.

3.3. Influence of the Bloom Filters Release on the False Positive Rate

As said before, when a new signature is needed and there is not room to host it, the controller must free some Bloom filters assigned to other signatures and assign them to the new signature. But, how does this affect the false positive rate?

The probability of a false positive is $P = (1 - (1 - k/m)^n)^k$ (see section 2.2), being m the number of bits of the signature, k the number of registers or hash functions, and n the number of elements inserted in the signature. In *FlexSig*, the relation m/k is constant. When the number of Bloom filters assigned to a signature is reduced, m and k are reduced in the same proportion.

Let us illustrate this influence with an example. Figure 7 shows the variation of the false positive rate when the resources allocated to a given signature are reduced: as an instance, assume that initially the signature is composed of $k = 16$ filters with a total register size of $m = 2048$ and it is reduced to $k = 8$ filters with $m = 1024$. If the number of addresses inserted in the signature, n, is low, the reduction in signature size has no practial influence on the false positive rate. However, if n is large the false positive rate increase significantly.

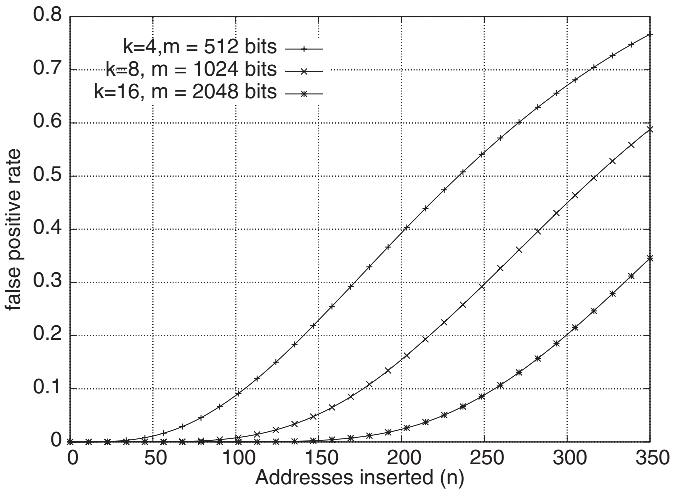


Fig. 7. False positive rate in *FlexSig* for different signature sizes.

3.4. Software Interface

To provide a basic software interface to *FlexSig*, the following instruction set extensions should be added.

—*Allocate(ID)*: allocate the signature with identifier ID. The size of the allocated signature is defined by the controller.
—*Deallocate(ID)*: deallocate the signature with identifier ID.
—*Insert(addr, ID)*: insert the address *addr* in the signature with identifier ID.
—*Check(addr, ID)*: check if address *addr* was inserted previously in the signature with identifier ID.

This basic instruction set extension allows the proper interface to the basic operations of the module. Moreover, the instruction set can be extended with new instructions for specific purposes. For instance, to support Transactional Memory, it can be extended with functionalities to forward signatures to cores, etc.

3.5. Register Grouping

In the case of just one or few signatures allocated in *FlexSig*, the number of Bloom filter elements per signature is high, i.e., equivalent to having a high value of k in conventional parallel Bloom filters. However, as we learned from Figure 2(b), the optimum value of k, in terms of the false positive rate, is low (between 3 and 8 for the parameters used in Figure 2(b)). To reduce the false positive rate in these cases, *FlexSig* can group several Bloom filters so that only one is used at a time in operations that involve hash functions (insert and check). A simple implementation consists of selecting the Bloom filter of the group based on the value of a few least significant bits of the address involved in the operation. Figure 8 illustrates the grouping scheme for groups of two elements.

Grouping can be implemented in a static or dynamic way. For a static implementation, the grouping size is chosen before the first allocation, and only can change when the *FlexSig* is totally empty. This forces all signatures to have the same grouping size and simplify the implementation. If it is implemented dynamically, the grouping size is chosen for each signature when is allocated, depending on the number of Bloom filters

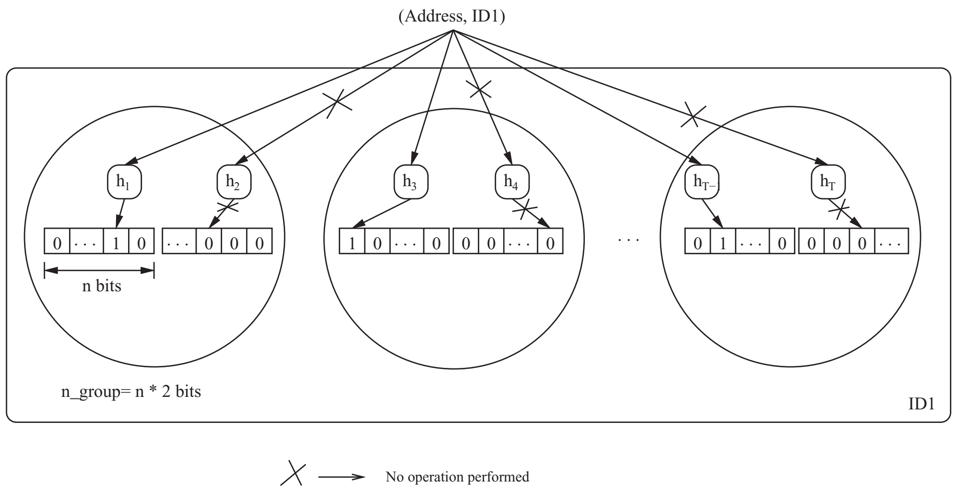


Fig. 8. Example of register grouping, for the case of only one signature allocated and grouping of two elements.

assigned to it. This complicates the logic and can cause other problems with many corner cases. Because of that, we chose to implement static grouping in our evaluation.

The maximum level of grouping is a design decision for *FlexSig*. The specific grouping for each application can be established through the software interface. For the case of our benchmarks (see Section 5.3) with up to 16 threads, we determined that a maximum grouping of two elements is enough to achieve good results. Moreover, this grouping is activated only for applications configured to run with two and four threads.

3.6. *FlexSig* Overflow and Fault Tolerance

FlexSig allows to host several signatures at the same time. However, a situation of overflow may be produced in exceptional (low probability) cases when a new allocation request arrives, and the controller can not free any Bloom filter because *FlexSig* hits the maximum number of signatures allowed (when the software tries to allocate more signatures than the total number of Bloom Filters available). In this case, the situation is managed by software, as it is done, for instance, for conventional signatures in Transactional Memory implementations.

In the case that an application allocates signatures, but fails to deallocate them (due to a software bug or fault), *FlexSig* will have fewer resources to allocate new signatures for the remaining running application time (similar to the memory leak problem). This case is very hard to manage in hardware, and therefore it should be handled by software. As an instance, a straightforward scheme for Transactional Memory is to clear *FlexSig* when serial code is executing or when no transactions are running in the system.

FlexSig has nice fault tolerant properties regarding the storage of the signatures due to its flexibility. If one register fails (permanent or soft error detected with standard fault detection techniques), such register is marked as invalid if the error is permanent (not used any more) or freed if it is a soft error (it can be used in new signature allocations). Moreover, regarding permanent faults, only stuck-at-zero faults would lead to the invalidation of a register (stuck-at-one faults only increase the false positive rate). No special operations are needed for managing this situation, as the only implication of loosing one Bloom filter is to increase the false positive rate. Of course, an exception is raised if the failing Bloom filter is the only one assigned to the signature.

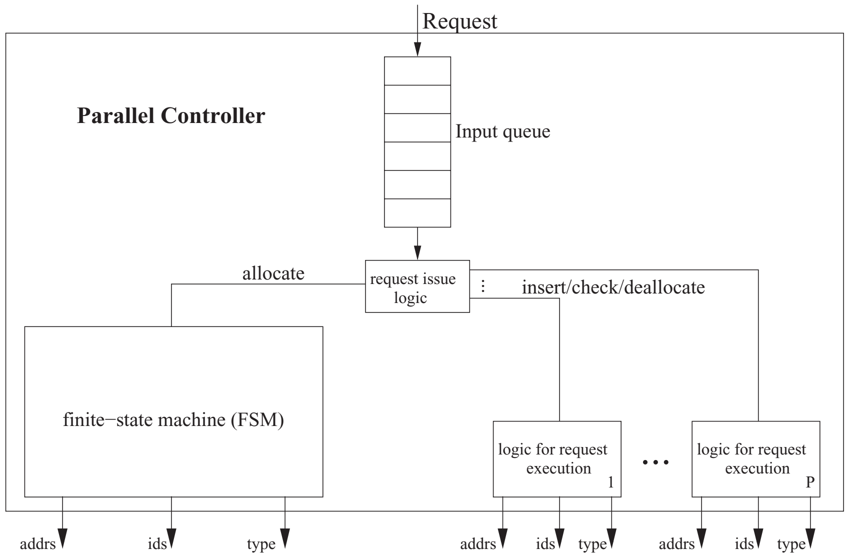


Fig. 9. Parallel controller implementation.

4. IMPLEMENTATION ISSUES

The *FlexSig* system can be placed in each core or as a centralized resource attached to the chip interconnection network. The flexibility of *FlexSig* is achieved at the cost of extra logic compared with conventional parallel Bloom filters. In this sense, a key element is the *FlexSig* controller, which should support the functionality of *FlexSig* with a simple architecture to reduce power an area overhead.

The controller needs one queue for the incoming requests, because the requests are served sequentially. There are four types of request, *Allocate*, *Deallocate*, *Check* and *Insert*, each with an ID that the controller uses to take action on the corresponding registers. To implement efficiently the straightforward allocation algorithm described in Section 3.2, some extra registers are needed in the controller to take fast decisions for the allocation operation. There are T $\log_2(T)$-bit counters in the controller to count the number of registers of *FlexSig* allocated by the corresponding ID. There is also a counter that keeps track of empty records. Using this stored information a finite-state machine performs the allocation operations.

For implementations using a centralized *FlexSig* for all cores, the module might be a bottleneck. After analyzing the concurrency of the possible arriving requests, we determined that the controller can serve some of them in parallel.

—*Insert*: It can be executed in parallel with other *inserts*, *checks* or *deallocates* with different IDs.
—*Check*: It can be executed in parallel with other *inserts* and *deallocates* with different IDs and with other *checks* with any ID.
—*Allocate*: It can not be executed in parallel with other requests.
—*Deallocate*: It can be executed in parallel with *deallocates*, *inserts* and *checks* with different ID.

Taking into account these rules, the controller can be parallelized in several ways. Figure 9 shows a simple parallel controller proposal. This controller may perform up to P operations in parallel. Basically it is an in-order issue superscalar engine. The incoming requests are placed in an input queue. The issue logic determines up to P requests to be issued in parallel, following the rules listed above. We have one finite-state machine (and the corresponding counters) to execute the *allocate* requests, and P very simple circuits to process *inserts*, *checks* or *deallocates*.

Most of the time, the finite-state machine has the calculations ready when a new *allocate* request arrives, because it recalculates these parameters immediately after the previous allocate or deallocate request. Only when two consecutive allocate requests arrive, the finite-state machine has no time to recalculate the parameters before the second request, incurring in some additional delay.

5. EVALUATION IN A TRANSACTIONAL MEMORY SYSTEM

The aim of the evaluation is to show the effectiveness of the *FlexSig* system with respect to conventional parallel Bloom filters. In this work we concentrate on Transactional Memory applications, since for many Transactional Memory implementations signatures are a key element. Transactional Memory uses signatures to detect conflicts among transactions. Each transaction inserts in signatures its reads and writes to maintain a summary of its read/write set. Conflicts with other transactions reads/writes are detected through the check operation. Since our purpose is evaluate only signatures, our figures of merit are in terms of false positive rates. A higher the false positive rate degrades performance, because for each false positive, the Transactional Memory system has to do an unnecessary abort (rollback to the initial state and restart the transaction).

To evaluate *FlexSig*, we use unified signatures (see Section 5.1), so we only need one signature per transaction for the read and write set, which allows us to implement the simple allocation algorithm described in Section 3.2. The results we obtained with our experimental setup (see below) using separate signatures are worse, and therefore we only report the results for unified signatures.

5.1. Unified Signatures: Simplifying *FlexSig* Implementation in Transactional Memory

Transactional Memory uses two signatures per transaction, one for the read set and another one for the write set. Usually the read set is larger than the write set, and therefore, in order to use efficiently the resources, the signature of the read set should be larger than the signature of the write set. However, having signatures of different sizes for the write set and the read set introduces additional difficulties in the allocation algorithm and makes its implementation more complex. Unified signatures [Choi and Draper 2011] propose to use only one signature for both the read set and the write set. This approach may generate read-read conflicts, however, these conflicts rarely lead to a performance lost [Sanyal et al. 2009; Choi and Draper 2011]. Using unified signatures each thread only needs to allocate one signature per transaction, and the complexity of the controller is reduced. This is the approach we have used for evaluating *FlexSig*.

5.2. Experimental Setup

To evaluate the *FlexSig* scheme we use a Transactional Memory system with signatures used to track data accesses in transactions. Our aim is not to implement a fully functional Transactional Memory system, but to work out a challenging scenario for *FlexSig*, and compare it with conventional parallel Bloom filters in the same situation. For the Transactional Memory system we use the software approach RSTM [Spear et al. 2008]. RSTM is a software Transactional Memory system that allows many different configurations. In our evaluation we use a lazy acquisition and lazy versioning with extendable timestamps [Riegel et al. 2007] to configure RSTM. We use PIN [keung Luk et al. 2005] to track all transactions and memory accesses of RSTM and to emulate the

Table I. Benchmark Inputs

Bench.	input
genome	-g128
intruder	-a10 -l16 -n4096 -s1
kmeans-high	-m15 -n15 -t0.05 -i random-n2048-d16-c16.txt
kmeans-low	-m40 -n40 -t0.05 -i random-n2048-d16-c16.txt
labyrinth	-i random-x256-y256-z3-n256.txt
ssca	-s14 -i1.0 -u1.0 -l9 -p9
vacation-high	-n4 -q60 -u90 -r1048576 -t4096
vacation-low	-n2 -q90 -u98 -r1048576 -t4096
yada	-a10 -i ttimeu10000.2
streamcluster	10 20 32 4096 4096 1000
canneal	2000 2000 10.nets

Table II. Benchmark Set A, Characterization with 16 Threads

Benchmark	#Tx	TxTime	RS	WS
intruder	101780	32%	19	2
vacation-high	4096	94%	384	7
vacation-low	4096	94%	283	5
yada	14316	68%	59	17
LinkedList	175	52%	141	0.3
DList	152	55%	138	0.6
PrivList	94	81%	256	1

hardware signatures. This conforms the simulation of a Hybrid Transactional Memory system.

We run several benchmarks over the RSTM system. Specifically, we use all the STAMP Benchmarks [Cao Minh et al. 2008], two PARSEC Benchmarks [Bienia et al. 2008] and nine micro benchmarks (included in the RSTM distribution). Table I shows the inputs of the benchmarks. The benchmarks not included in the table run with the default input. For this evaluation, we classify the benchmarks in two categories. One group is composed of benchmarks with a high false positive rate (Benchmark set A), and the other with a modest false positive rate (Benchmark set B). The purpose of this is to run each group of benchmarks with a different signature configuration to show the advantages of *FlexSig* for workloads with different characteristics.

Tables II and III show the characterization of the benchmarks. The parameter $\#Tx$ is the number of transactions of the benchmark, $TxTime$ is the percentage of time spent on transactions, and RS and WS are the average number of reads and writes per transaction. The time spent in transactions is, in general, very significant. This parameter is affected by the instrumentation tool, because only transactions are instrumented. This scenario is a pessimistic approximation, since in a real system the time spent inside the transactions should be less, and therefore, it should be less likely that those transactions demand signatures at the same time in the *FlexSig* system. Therefore, the results should be better than in the simulated case.

5.3. Configuration

For the evaluation we use the configurations shown in Table IV. The hardware configuration for parallel Bloom filters (k and m are the parameters in Figure 1(b)) was chosen specifically to manage up to 16 threads (that is, the conventional signature system has 16 parallel Bloom filters of fixed size). We run experiments with 2, 4, 8 and 16 threads. Two configurations are used for *FlexSig*: configuration *conf1* uses the same resources as their equivalent parallel Bloom filter, and *conf2* uses half of the resources. For the benchmarks belonging to the set A, the registers are of 512 bits for the unified parallel Bloom filter (a total of 8192 bits for a 16 thread system); for *FlexSig* we have

Table III. Benchmark Set B, Characterization with 16 Threads

Benchmark	#Tx	TxTime	RS	WS
bayes	644	46%	8	2
genome	353994	76%	26	0
kmeans-high	8238	43%	13	13
kmeans-low	8557	70%	13	13
labyrinth	544	54%	84	80
ssca	93731	49%	1	2
streamcluster	592	17%	1	0
canneal	4000	44%	2	1
Counter	759	23%	1	1
HashTable	2772	47%	2	0.3
RBTree	16385	68%	18	2
RBTreeLarge	134	61%	27	3
LFUCache	62	61%	7	2
RandomGraph	53	59%	506	2

Table IV. Configuration Used with Unified Signatures

Signature	Description
Unified Parallel Bloom (set B)	16 registers with k=4 and m=32 bits (512 bits total)
Unified Parallel Bloom (set A)	16 registers with k=4 and m=512 bits (8192 bits total)
Unified *FlexSig* conf2 (set B)	32 registers of 8 bits (256 bits total)
Unified *FlexSig* conf1 (set B)	64 registers of 8 bits (512 bits total)
Unified *FlexSig* conf2 (set A)	32 registers of 128 bits (4096 bits total)
Unified *FlexSig* conf1 (set A)	64 registers of 128 bits (8192 bits total)

Table V. Benchmark Set A. False Positives Comparison (in %) for Unified Signatures

Benchmark	2 threads			4 threads			8 threads			16 threads		
	Bloom	conf2	conf1	Bloom	conf2	conf1	Bloom	conf2	conf1	Bloom	conf2	conf1
intruder	1.5	0.0	0.0	2.0	0.2	0.0	2.4	2.4	0.2	2.9	12.0	2.7
vacation-high	38.1	3.7	0.5	37.9	13.0	3.7	38.1	38.0	24.2	38.3	52.9	38.2
vacation-low	25.3	0.8	0.0	25.3	6.0	0.8	25.4	25.3	11.1	25.2	42.5	25.1
yada	18.8	0.6	0.0	19.7	4.7	0.7	20.4	20.4	8.9	20.1	34.4	20.1
LinkedList	4.6	0.1	0.0	2.6	0.4	0.0	1.5	1.5	0.2	0.7	4.7	0.7
DList	4.0	0.1	0.0	2.9	0.5	0.0	2.0	2.0	0.2	0.7	2.9	0.7
PrivList	6.2	0.5	0.1	8.1	3.8	1.5	3.5	3.5	2.1	4.2	7.1	4.2

32 registers of 128 bits for *conf2*, and 64 registers of 128 bits for *conf1*. Similarly, for the benchmarks belonging to the set B, the size of the registers for the unified parallel Bloom filter is 32 bits and the corresponding *FlexSig* configurations *conf1* and *conf2* are described in Table IV. To group registers (see Section 3.5), we choose groups of one register for 8 and 16 threads, and groups of two registers for executions with 2 and 4 threads. This decision was taken to have an efficient configuration (see Figure 2(b)).

5.4. Results

Tables V and VI show the false positive rate of *FlexSig* with configurations *conf1* and *conf2* compared with the results obtained with parallel Bloom filters (see Table IV), for the case of 2, 4, 8 and 16 threads. A white cell (in *conf1* and *conf2* columns) means that the false positive rate is roughly the same as the one obtained with the parallel Bloom filter, a gray cell means that the false positive rate of *FlexSig* is better (lower), and a dark gray means that the false positive rate of *FlexSig* is worse (higher).

First, we comment the results with *conf1* for both benchmark sets A and B. As Tables V and VI show, the *FlexSig-conf1* outperforms parallel Bloom filters in almost all the cases. For 2, 4 and 8 threads, the improvement is very high; for instance, for the case of *vacation-high* running with 2 threads, the false positive rate is reduced

Table VI. Benchmark Set B. False Positives Comparison (in %) for Unified Signatures

Benchmark	2 threads			4 threads			8 threads			16 threads		
	Bloom	conf2	conf1	Bloom	conf2	conf1	Bloom	conf2	conf1	Bloom	conf2	conf1
bayes	43.2	7.1	3.0	33.1	13.0	6.5	32.4	31.7	23.5	24.9	34.1	24.4
genome	41.8	4.8	0.7	40.7	13.6	4.6	43.3	42.0	29.2	9.7	54.4	9.4
kmeans-low	45.1	11.2	1.0	41.2	17.9	10.6	40.0	40.0	36.4	36.7	69.8	36.6
kmeans-high	40.6	7.6	0.7	37.6	16.4	8.3	36.6	36.6	33.8	37.5	66.1	37.4
labyrinth	19.0	17.9	14.7	16.7	15.8	15.6	41.9	41.9	41.6	72.2	77.3	72.2
ssca	1.1	0.0	0.0	1.1	0.1	0.0	1.2	1.1	0.0	1.2	7.7	1.1
streamcluster	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
canneal	3.3	0.0	0.0	3.2	0.0	0.0	3.5	3.5	0.5	3.3	9.6	3.1
Counter	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
HashTable	2.5	0.0	0.0	2.0	0.1	0.0	1.9	1.9	0.2	1.9	8.8	1.7
RBTree	49.7	16.7	8.3	53.6	30.8	19.7	46.8	46.8	40.3	44.0	50.1	44.0
RBTreeLarge	48.8	16.5	8.3	47.1	25.1	16.3	44.5	44.5	35.8	37.4	46.4	37.3
LFUCache	26.2	8.8	4.7	28.3	12.8	10.5	29.7	29.7	26.2	27.6	31.6	27.6
RandomGraph	17.9	13.4	11.8	17.0	12.7	10.7	16.6	16.6	14.4	16.9	20.3	16.9

from 38,1% to 0,5%. As the number of threads increases, the advantage of *FlexSig* decreases. However, even in the worst case (16 threads), *FlexSig* improves with respect to conventional Bloom filters in many cases, and never performs worse. The results are better as fewer threads are running, because *FlexSig* tries to use all the resources, while the parallel Bloom filter implementation has fixed size for each signature independently of the number of threads. The results of *FlexSig-conf1* with 16 threads are very similar to the implementation with parallel Bloom filters since for this case all the signatures are used. *FlexSig* achieves better results because not all the threads allocate signatures at the same time, and it can use the free resources also in this case. The results are only slightly better because the benchmarks are highly concurrent (in part due to the instrumentation performed by PIN).

FlexSig-conf2 uses half of the resources of the parallel Bloom filter implementation. Even with this configuration, *FlexSig* clearly outperforms the parallel Bloom filter implementation for 2 and 4 threads. For instance, for *vacation-high* the false positive rate with two threads is reduced from 38,1% to 3,7%. For the case of 8 threads, the results are similar in both implementations, but *FlexSig* outperforms the parallel Bloom filter implementation in many cases, and at least matches it. For 16 threads, *FlexSig* has worse performance, but it has the flexibility to manage the 16 threads with half the resources.

It is of interest to have an estimation of the average signature size that is used per transaction in *FlexSig*. The average signature size is the weighted average in time of the signature size, and is given by

$$ave\_sig\_size = \frac{\sum_{e=1}^{num\_changes\_size} sig\_size_e * time\_interval_e}{\sum_{e=1}^{num\_changes\_size} time\_interval_e} \qquad (2)$$

where *num_changes_size* is the number of times a signature changes its size (number of registers) before deallocation. *time_interval* is the number of time units that a signature has a size *sig_size*.

Figure 10 shows the improvement in the average signature size of *FlexSig-conf1* compared with the equivalent conventional signature. This improvement is achieved because not all the threads use signatures simultaneously, and therefore, the threads can take resources that others threads do not use. The signature size for *FlexSig* depends basically on the concurrent nature of the benchmark (less concurrent threads lead to a better performance of *FlexSig*). In a similar way, Figure 11 shows the average signature size improvement for *FlexSig-conf2* compared with the equivalent

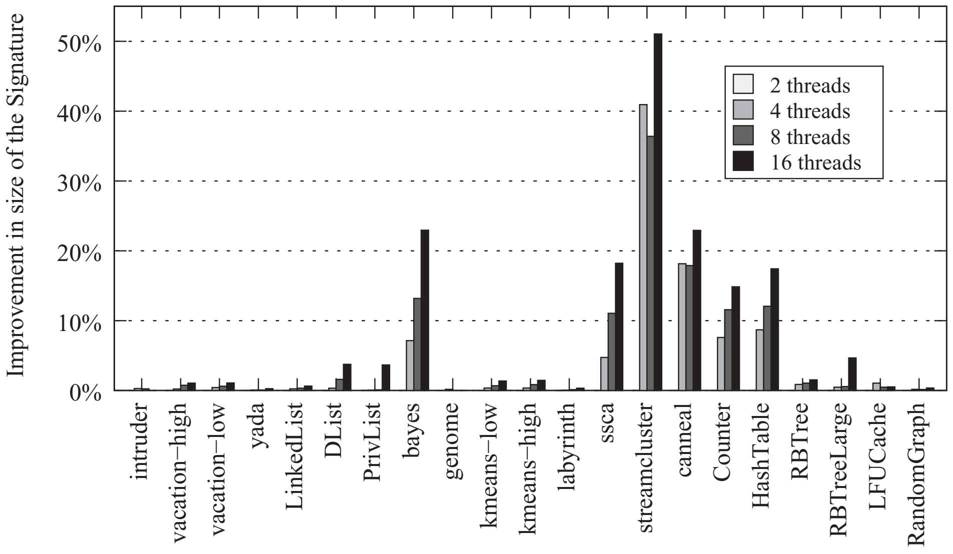


Fig. 10. Increment of the average signature size in *FlexSig-conf1* compared with regular signatures. The size in *FlexSig* is calculated with the per transaction average size of signature.

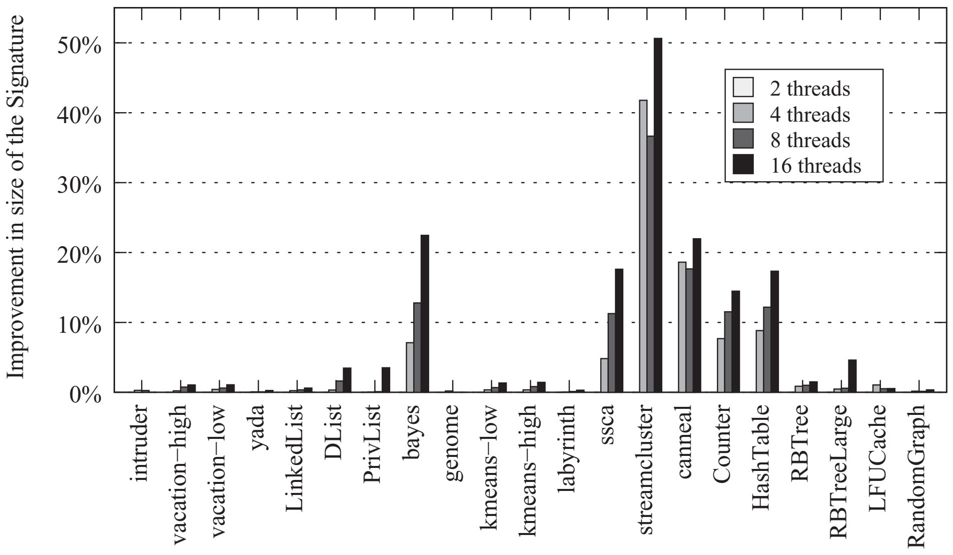


Fig. 11. Increment of the average signature size in *FlexSig-conf2* compared with regular signatures.

conventional signatures. Despite the fact that the resources are a half of the *conf1*, the improvement achieved shows a similar behavior (Figure 10). The best result in terms of *FlexSig* average signature size improvement is for *streamcluster*, with a more than 50% improvement, due to the low transaction concurrency in this benchmark. In this case, the improvement of the signature size doesn't imply a significant reduction of the false positive rate because this is already very low in absolute terms.

As a conclusion, the *FlexSig* system improves the false positive rate when compared with conventional parallel Bloom filters. In a system configured to run 16 threads, our signature system clearly outperforms the parallel Bloom filter implementation when the number of threads is lower than 16 (for the *conf1* with the same resources), due to the flexibility of *FlexSig* to assign the physical registers depending on the demand (number of concurrent threads). General purpose multicore and multiprocessors are able to run a large number of concurrent threads, but many applications use only a few threads. *FlexSig* is flexible enough to provide these applications all the available signature resources to achieve better performance. We used very hard conditions in our evaluation to demonstrate that *FlexSig* can perform well even in an unfavorable scenario. The benchmarks used are highly concurrent, which means that many transactions use signatures at the same time. Moreover, because of the instrumentation tool, the benchmarks spend more time inside transactions, increasing transactional concurrency.

6. RELATED WORK

Most of the papers dealing with signatures are focused on improving performance, reducing chip area or reducing the false positive rate [Sanchez et al. 2007; Quislant et al. 2009; Yen 2009; Shenghua et al. 2009; Yen et al. 2008]. However, none of these papers focus on flexibility and scalability. The Scalable Bloom Filters (SBF) proposed by [Almeida et al. 2007] tries to make an approximation of scalable signatures. They use one signature, and when a fill ratio is reached, another signature is used. SBF was proposed to avoid the problem of oversize signatures due to the fact that the size of the signature must be defined previously based on the number of elements to be stored and the desired upper bound of the false positive rate. The SBF method can reduce the specific size of the signature used. However, it may use several signatures depending on the number of elements to be stored, and therefore, in reference to our context, the system has to be oversized anyway (with regard to the number of signatures). *FlexSig* does not fully avoid the problem of oversized signatures, but it is more flexible and efficient in the sense that it uses as many hardware resources as possible, having a significant effect on the false positive rate for a Transactional Memory System.

In recent publications we find Transactional Memory systems that fit very well for using *FlexSig*. Mehrara et al. [2009] proposes a Software Transactional Memory system with a centralized conflict detection mechanism (based on software signatures) placed in one core. One way to improve this scheme would be to use *FlexSig* instead of their software signatures. This would improve performance maintaining the flexibility of the software signatures. Another example is the scheme proposed by Casper et al. [2011], that describes a new centralized hardware outside the processor chip to accelerate Software Transactional Memory systems. This special hardware includes signatures. They also propose two algorithms for conflict detection, one using two signatures per transaction and other using three signatures. *FlexSig* would allow to implement both with the same hardware and also will improve performance.

The idea behind *FlexSig* is similar to the recent trend of incorporating a shared last level cache in multicore systems. The cache size used by each core varies dynamically depending on the application. This leads to a more flexible system than having a fixed size slice of the last level cache assigned to each core. *FlexSig* follows this trend for a resource that might be of interest for future multicore implementations.

7. CONCLUSIONS

In this work we propose a module for hardware signatures to improve conventional signatures in terms of flexibility, scalability and fault tolerance. The main feature of *FlexSig* is that it can host a high number of signatures for cases with applications with a high number of threads and significant contention, and for the cases for low contention or few threads, it can achieve a very low false positive rate.

We described the module and its implementation, defined a detailed algorithm to allocate signatures and evaluated *FlexSig* in the context of a Transactional Memory system and compared it to an implementation with conventional parallel Bloom filters. From the evaluation performed, we show that, when the number of threads is low, *FlexSig* achieves a significant improvement because of the flexibility to use all the available resources. When the number of threads is high, the results are similar to the conventional implementation due to the highly concurrent nature of the benchmarks. However, with the same amount of resources, *FlexSig* never behave worse than conventional parallel Bloom filters.

FlexSig makes signatures more flexible to use as a general purpose hardware resource, since it is able to adapt to the concurrent demand of signatures, and decouples, to some extent, the type of benchmark from the hardware.

As future work, we will explore more complicated allocation algorithms using priorities and will test the efficiency of *FlexSig* in other environments outside Transactional Memory.

REFERENCES

Adve, S. V., Hill, M. D., Miller, B. P., and Netzer, R. H. B. 1991. Detecting data races on weak memory systems. In *Proceedings of the 18th Annual International Symposium on Computer Architecture. (ISCA '91)*. ACM, New York, NY, 234–243.

Almeida, P., Baquero, C., Preguica, N., and Hutchison, D. 2007. Scalable bloom filters. *Infor. Process. Lett. 101,* 6, 255–261.

Ananian, C. S., Asanovic, K., Kuszmaul, B. C., Leiserson, C. E., and Lie, S. 2005. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, Los Alamitos, CA, 316–327.

Bienia, C., Kumar, S., Singh, J. P., and Li, K. 2008. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, (PACT '08)*. ACM, New York, NY, 72–81.

Bloom, B. H. 1970. Space/time trade-offs in hash coding with allowable errors. *Comm. ACM 13*, 422–426.

Cao Minh, C., Chung, J., Kozyrakis, C., and Olukotun, K. 2008. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of the IEEE International Symposium on Workload Characterization*.

Carter, J. L. and Wegman, M. N. 1977. Universal classes of hash functions (extended abstract). In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, (STOC '77)*. ACM, New York, NY, 106–112.

Casper, J., Oguntebi, T., Hong, S., Bronson, N. G., Kozyrakis, C., and Olukotun, K. 2011. Hardware acceleration of transactional memory on commodity systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, (ASPLOS '11)*. ACM, New York, NY, 27–38.

Ceze, L., Tuck, J., Montesinos, P., and Torrellas, J. 2007. Bulksc: bulk enforcement of sequential consistency. In *Proceedings of ISCA*.

Ceze, L., Tuck, J., Torrellas, J., and Cascaval, C. 2006. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*. IEEE Computer Society, 238.

Chang, F., chang Feng, W., and Li, K. 2004. Approximate caches for packet classification. In *INFOCOM 2004. Proceedings of the 23th Annual Joint Conference of the IEEE Computer and Communications Societies*. Vol. 4, 2196–2207.

Chang, F., Li, K., and chang Feng, W. 2004. Approximate caches for packet classification. In *Proceedings of INFOCOM*.

Choi, J.-D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., and Sridharan, M. 2002. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI '02)*. ACM, New York, NY, 258–269.

Choi, W. and Draper, J. 2011. Implementation of unified signatures for transactional memory systems. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*.

Fan, L., Cao, P., Almeida, J., and Broder, A. Z. 2000. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw. 8*, 281–293.

HAMMOND, L., WONG, V., CHEN, M., CARLSTROM, B. D., DAVIS, J. D., HERTZBERG, B., PRABHU, M. K., WIJAYA, H., KOZYRAKIS, C., AND OLUKOTUN, K. 2004. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture, (ISCA '04)*. IEEE Computer Society, Los Alamitos, CA, 102.

HERLIHY, M. AND MOSS, J. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. ACM, 300.

KEUNG LUK, C., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., JANAPA, V., AND HAZELWOOD, R. K. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*. ACM, 190–200.

LEV, Y. AND MOIR, M. 2006. Debugging with transactional memory. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*.

LUCIA, B., DEVIETTI, J., STRAUSS, K., AND CEZE, L. 2008. Atom-aid: Detecting and surviving atomicity violations. In *Proceedings of the 35th Annual International Symposium on Computer Architecture, (ISCA '08)*. IEEE Computer Society, Los Alamitos, CA, 277–288.

MEHRARA, M., HAO, J., HSU, P.-C., AND MAHLKE, S. 2009. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, (PLDI '09). ACM, New York, NY, 166–176.

MUZAHID, A., SUÁREZ, D., QI, S., AND TORRELLAS, J. 2009. Sigrace: signature-based data race detection. In *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*. ACM, New York, NY, 337–348.

NETZER, R. H. B. AND MILLER, B. P. 1991. Improving the accuracy of data race detection. In *Proceedings of the 3th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPOPP '91)*. ACM, New York, NY, 133–144.

NETZER, R. N. AND MILLER, B. P. 1989. Detecting data races in parallel program executions. In *Proceedings of the 1990 Workshop on Advances in Languages and Compilers for Parallel Computing*. MIT Press, 109–129.

O'CALLAHAN, R. AND CHOI, J.-D. 2003. Hybrid dynamic data race detection. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPoPP '03)*. ACM, New York, NY, 167–178.

PRVULOVIC, M. AND TORRELLAS, J. 2003. Reenact: using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the 30th Annual International Symposium on Computer Architecture, (ISCA '03)*. ACM, New York, NY, 110–121.

QUISLANT, R., GUTIERREZ, E., PLATA, O., AND ZAPATA, E. L. 2009. Improving signatures by locality exploitation for transactional memory. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, Los Alamitos, CA, 303–312.

RAMAKRISHNA, M. V., FU, E., AND BAHCEKAPILI, E. 1997. Efficient hardware hashing functions for high performance computers. *IEEE Trans. Comput. 46*, 1378–1381.

RATANAWORABHAN, P., BURTSCHER, M., KIROVSKI, D., ZORN, B., NAGPAL, R., AND PATTABIRAMAN, K. 2009. Detecting and tolerating asymmetric races. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York, NY, 173–184.

REYNOLDS, P. AND VAHDAT, A. 2003. Efficient peer-to-peer keyword searching. In *Proceedings of the ACM/IFIP/USENIX International Conference on Middleware, (Middleware '03)*. Springer-Verlag, Berlin, 21–40.

RHEA, S. AND KUBIATOWICZ, J. 2002. Probabilistic location and routing. In *INFOCOM 2002. Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*. Vol. 3, 1248–1257.

RIEGEL, T., FETZER, C., AND FELBER, P. 2007. Time-based transactional memory with scalable time bases. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures, (SPAA '07)*. ACM, New York, NY, 221–228.

SANCHEZ, D., YEN, L., HILL, M. D., AND SANKARALINGAM, K. 2007. Implementing signatures for transactional memory. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, (MICRO 40)*. IEEE Computer Society, Los Alamitos, CA, 123–133.

SANYAL, S., ROY, S., CRISTAL, A., UNSAL, O. S., AND VALERO, M. 2009. Dynamically filtering thread-local variables in lazy-lazy hardware transactional memory. In *Proceedings of the 11th IEEE International Conference on High Performance Computing and Communications*. IEEE Computer Society, Los Alamitos, CA, 171–179.

Shenghua, Z., Zheng, Q., Yuan, Z., and Xiaolan, P. 2009. A cascade hash design of bloom filter for signature detection. In *Proceedings of the International Forum on Information Technology and Applications, (IFITA '09)*. Vol. 2, 559–562.

Spear, M. F., Michael, M. M., and von Praun, C. 2008. Ringstm: scalable transactions with a single atomic instruction. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures, (SPAA '08)*. ACM, New York, NY, 275–284.

Tuck, J., Ahn, W., Ceze, L., and Torrellas, J. 2008. Softsig: software-exposed hardware signatures for code analysis and optimization. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, (ASPLOS XIII)*. ACM, New York, NY, 145–156.

Yen, L. 2009. Signatures in transactional memory systems. Ph.D. thesis, Madison, WI.

Yen, L., Bobba, J., Marty, M. R., Moore, K. E., Volos, H., Hill, M. D., Swift, M. M., and Wood, D. A. 2007. Logtm-se: Decoupling hardware transactional memory from caches. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE Computer Society, Los Alamitos, CA, 261–272.

Yen, L., Draper, S. C., and Hill, M. D. 2008. Notary: Hardware techniques to enhance signatures. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture, (MICRO 41)*. IEEE Computer Society, Los Alamitos, CA, 234–245.

Yu, Y., Rodeheffer, T., and Chen, W. 2005. Racetrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles, (SOSP '05)*. ACM, New York, NY, 221–234.

Zhou, P., Teodorescu, R., and Zhou, Y. 2007. Hard: Hardware-assisted lockset-based race detection. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE Computer Society, Los Alamitos, CA, 121–132.

Received July 2011; revised October 2011; accepted November 2011