Advance Access publication on 17 March 2016
doi:10.1093/comjnl/bxw010

Asymmetric Allocation in a Shared Flexible Signature Module for Multicore Processors

Lois Orosa[1]*, Javier D. Bruguera[2] and Elisardo Antelo[3]

[1]*Institute of Computing, University of Campinas, Campinas, Brazil*
[2]*ARM, Cambridge, UK*
[3]*Department of Electronics and Computer Science, University of Santiago de Compostela, Santiago de Compostela, Spain*
**Corresponding author: lois.orosa@ic.unicamp.br*

Hardware signatures based on Bloom filters are used to support and accelerate membership query in a set of items. They use modest hardware at the cost of false positives, but never produce false negatives. Signatures were traditionally used in different distributed and network applications, but in recent years their use has been extended to other fields (for instance, support for manycore/multicore parallel programming, such as data race detection, deterministic replay or transactional memory (TM)). One drawback of signatures is that they have a fixed size, and what is a good signature size for one application, may be not appropriate for another. Recently, we proposed a shared hardware module for managing signatures based on a collection of Bloom filters. It has the characteristic of hosting a variable number of signatures that change their size in runtime to adapt to the demand of the applications. However, the assignment of resources follows a single symmetric policy for all allocations leading to a module with a limited adaptability to the workloads. In this paper, we explore new techniques to allocate signatures in an asymmetric way in this module, with the aim of optimizing the resources and reducing even more the number of false positives. We explore several asymmetric strategies and their efficient hardware implementation, and we show specific examples using TM as a driver application. The results show that these strategies lead to a significant reduction in the number of false positives compared with symmetric policies.

Keywords: signatures; Bloom filters; multithreaded applications; multicore architectures

Received 31 March 2015; revised 29 November 2015
Handling editor: Alan Marshall

1. INTRODUCTION

Bloom filters [1] are a probabilistic data structure for membership query that are space-efficient at the cost of producing false positives (but never false negatives), and its design space introduces a trade-off between the false positive rate and the filter complexity. The information stored in the Bloom filter is a signature of a group of data items, and it is used for membership query in this set when the application tolerates false positives. Subsequent variants of Bloom filters appeared to cover the needs of new applications, some of them can even report false negatives [2–5], but in this paper we refer to this standard definition. Bloom filters are widely used for distributed and network applications such as caching, database applications, peer to peer systems, routing and forwarding, and monitoring and measurement [6], and other applications in the fields of bioinformatics, big data and hardware support for multicore/manycore parallel programming [7]. A simple query in Google Scholar reveals and increasing variety of applications for Bloom filters.

Traditionally, the implementations for handling signatures with Bloom filters are done by software using the resources of a general purpose microprocessor. However, in recent years there have been several proposals for a dedicated hardware implementation, that we call hardware signatures. Specific implementations of hardware signature support for multicore/manycore parallel programming and hardware acceleration of other applications include among others: transactional

memory (TM) [8–12], data race detection and optimization [13, 14], efficient and safe shared memory execution [15] string matching [16], error detection and correction [17, 18], virus scanning [19], longest prefix matching [20], semantic information filtering [21], word matching for sequence analysis [22], intelligent DRAM refresh [23], energy-efficient processors for sensor networks [24] etc.

Multicore/manycore processors are the new driver of performance scaling by industry and a topic of intense research by academia. The energy and power wall is demanding energy-efficient hardware accelerator modules to support efficiently important sets of applications. Energy-efficient hardware accelerators may reduce energy and power by up to two orders of magnitude compared with general purpose hardware [25]. In this work, we argue that Bloom filters are a excellent candidate for a hardware accelerator module in multicore/manycore processors. Many applications in different fields that require probabilistic membership queries over large data items (tolerating certain false positive rate) could take advantage of this specific hardware for improving performance while reducing energy and power.

For accelerators to be useful, they must be flexible and adaptable to the workloads requirements. One important issue with hardware signatures is the correlation of the false positive rate with the size of the signature, the number of items inserted and the nature and statistical distribution of the items and queries. False positives might be tolerated by an application, but may lead to performance degradations if the false positive rate is too high. Therefore, a fixed static dimensioning of the hardware signatures for different applications may not be efficient. Following this line, in a recent work we proposed *FlexSig* (Flexible Signatures) [26], our first approach for a shared hardware signature module. *FlexSig* is a module that is able to host a variable number of signatures of variable size with an efficient and flexible use of the available hardware resources. *FlexSig* uses all the resources available for the allocated signatures when it is possible. The scheme resembles the concept of a shared last level cache for multiple cores. This flexibility allows to obtain fewer false positives in many applications and to adapt to applications and inputs of different characteristics.

A severe drawback of *FlexSig* is that it uses the same allocation policy for all the requested signatures (no priorities), which obviates that, in general, applications may have asymmetric characteristics (some require more resources than others). By analyzing these characteristics, and providing the appropriate allocation algorithm, it is possible to optimize the performance of *FlexSig*. This is in line with the fact that hardware accelerators should have enough flexibility to be used efficiently by a wide range of different applications. Another important drawback of *FlexSig* is that the allocation algorithm proposed in [26] might be slow for a high throughput system.

In this work, we go a step further for providing a flexible hardware signature module that incorporates policies of allocation based on priorities: *FlexSigP*. Specifically, we explore new asymmetric allocation algorithms and their efficient high throughput hardware implementation to reduce further the number of false positives by exploiting the asymmetry present in software applications. The driver application for testing our hardware proposal is TM [27–30]. Hardware TM is already included in restricted form in widely used Intel platforms [30] and a relevant hybrid (combined software and hardware) TM system for this platform has been recently proposed using software Bloom filters [31]. On the other hand, *FlexSigP* is a hardware accelerator that could be used for other applications as well by simply defining the appropriate ISA interface. *FlexSigP* achieves important reductions of false positives compared with *FlexSig* and conventional static hardware signatures. This reduction in the number of false positives may have a direct impact on the performance of the applications.

The rest of the paper is organized as follows. In Section 2, hardware signatures are briefly introduced. In Section 3, we review our previous work on *FlexSig*. Section 4 presents our new work on asymmetric allocation algorithms. Section 5 discusses an efficient hardware implementation. Related system issues are discussed in Section 6. We evaluate our work in Section 7 in the context of TM workloads, review related works in Section 8 and finally, Section 9 is devoted to the conclusions.

2. HARDWARE SIGNATURES

Without loss of generality, we discuss hardware signatures in the context of data items corresponding to memory addresses. Hardware signatures store an unbounded number of addresses in a bounded space. The most common type of signatures are Bloom signatures (implemented with Bloom filters [1]), which are composed of one or more hash operations (that encode the addresses) and one or more registers to store or check the bit positions that result from the hash operations. Although many variations of the basic Bloom filter have been proposed (Counting Bloom filter, Bloomier filters, etc, see [6] for an extended list), in this work we concentrate in the standard Bloom filter. However, the proposed module can be extended to other variations of Bloom filters, this requiring more research in order to design a reconfigurable hardware to support these variations.

There are two main operations in hardware signatures: to insert an address, and to check for membership. Since the storage space is limited, storing a high number of addresses in the signature increases the probability of aliasing. This affects the check operations, since the higher the aliasing, the higher the probability of reporting a false positive. On the other hand, signatures never report false negatives, that is, a negative result in the check operation is always correct. The main figure of merit of a signature is the false positive rate. There is a trade-off between the false positive rate, the size of the signature and the number of addresses inserted. As an instance, for a false positive rate of 1%, and n inserted addresses, a lower bound in the signature size required (m) is given by $m \approx 9.6n$ bits.

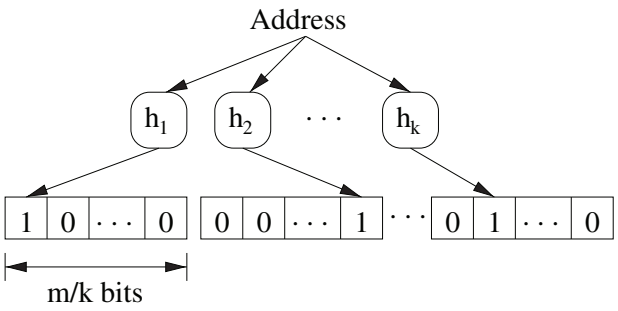

FIGURE 1. Implementation of a PBF.

There are two basic implementations of signatures based on standard Bloom filters. True Bloom signatures (implemented with single Bloom filters [1]) are composed of one multi-ported m-bit register, and by a set of k hash functions. Parallel Bloom signatures (implemented with parallel Bloom filters (PBFs) [12, 32]) are composed of a set of single-ported registers, and with one hash operation per register. PBFs achieve a false positive rate similar to True Bloom signatures, but are simpler to implement since costly multi-ported registers are avoided.

FlexSigP is based on PBFs. Figure 1 shows a generic PBF, composed of k hash–register pairs. The total effective size of the signature is m bits. Each hash–register pair can be seen as a single independent m/k-bit Bloom filter.

To insert an address in a PBF, each hash encodes the address, resulting in a specific bit position set to one in the register. To check for ownership, the hash encodes the address and checks if all the corresponding bits (the bit positions indicated by the hash operations) in the register are set.

3. FLEXIBLE HARDWARE SIGNATURES

FlexSig [26] is a new shared hardware signature module designed to make hardware signatures more flexible. The goal is to provide a shared resource for the threads in a multicore processor to make an efficient use of signatures as a supporting building block for parallel multicore applications. *FlexSig* allocates all the available resources dynamically, resulting of signatures of variable size.

FlexSig is based on PBFs, with a module composed of a set of True Bloom filters (BF) each one with one hash–register pair and an identifier that associates it with a signature. Therefore, *FlexSig* configures signatures of different sizes by grouping BFs with the same identifier.

FlexSig tries to use all the resources in the module to allocate the required number of signatures dynamically. The flexibility is achieved by reducing the size of the signatures dynamically if a new signature allocation request arrives. *FlexSig* frees BFs from signatures without concerns about correctness (false negatives are not produced), since reducing the size of a signature only affects the false positive rate. On the other hand, when *FlexSig* deallocates a signature and frees the corresponding BFs, these are not added to the remaining signatures due to the complexity involved.

In *FlexSig*, the symmetric allocation algorithm tries to assign the same number of BFs to all the signatures allocated in the module. The allocation policy proposed is very simple but slow. The controller computes the maximum number of BFs per signature, and then frees BFs in a round-robin fashion until the number of free BFs is equal to the maximum number of BFs per signature. This sequential process is done by a finite state machine and requires a variable (a potentially high) number of cycles. For a system with frequent allocations this results in a performance bottleneck. In this work, we propose a new high throughput allocation algorithm, extended to include also priorities, that is suitable for frequent allocations.

4. ASYMMETRIC ALLOCATION POLICIES

In this work, we propose *FlexSigP*, a module with new asymmetric allocation algorithms with a set of priorities to take advantage of asymmetric characteristics in applications or among applications. With an asymmetric strategy the controller assigns signatures with a different number of BFs depending on the priority. As we show in Section 7, the asymmetric policies lead to a significant reduction in the number of false positives compared with *FlexSig*.

The design space of *FlexSigP* is very wide and severely affects the hardware complexity of the allocator. Some of the parameters to consider are: number of states for a priority class, support for several orthogonal priority classes, variable or fixed ratio of resources in a priority class etc. After an exploration of the design space, we decided to use two orthogonal priority classes (signatures are allocated based on two independent priorities in a two level process), and each one with only two possible states: high priority and low priority. As we show in Section 5 these design decisions lead to a reasonable complexity increase in the allocator compared with the case without priorities, and it allows a significant reduction in the number of false positives (see Section 7). As we already achieve a good trade-off between complexity and performance, we do not consider more involved priority schemes which lead to very complex allocators.

4.1. High-level asymmetric allocation algorithm with two priority classes

In this section, we describe the high-level allocation algorithm with two orthogonal priority classes. Table 1 shows a summary of all the terms that we will use in our explanations. An agent (typically a thread) issues a request to allocate signatures. The outer priority class (PCOUT) determines the total number

TABLE 1. Summary of terms used in Section 4.

Name	Description
FlexSig	Symmetric signature module
FlexSigP	*FlexSig* with priorities (asymmetric)
PCOUT	outer priority class
PCIN	Inner priority class
P_H	Value coding the high priority
P_L	Value coding the low priority
S	Ratio of resources assigned
$N_L()$	Maximum number of BFs for a low-priority agent/signature
$N_H()$	Maximum number of BFs for a high-priority agent/signature
Q	Maximum number of BFs to be distributed
α_H	Number of agents/signatures of high priority
α_L	Number of agents/signatures of low priority
AS	Total number of signatures per agent
T	Total number of Bloom filters
F	Total number of agents

of BFs assigned to the agent. Regarding the PCOUT priority class, agents are assigned a high or low priority, and there can be any number of agents of different priority levels. The inner priority class (PCIN) is used to distribute the BFs assigned to the agent among its internal signatures (again with high or low priority).

Figure 2 shows a graphical example that illustrates PCIN and PCOUT. The PCOUT associates 6 BFs to the Agent1 (low priority), and 10 BFs to the Agent2 (high priority). The PCIN associates 3 BFs to Sig1 and 3 BFs to Sig2 (the priorities are the same, symmetric allocation) in Agent1, and 2BFs to Sig1 (low priority) and 8BFs to Sig2 (high priority) in Agent2.

We code the priority levels and the ratio of resources as follows. Let P_H (P_L) the value coding the high priority (low priority). The ratio of resources assigned, S, is given by

$$S = \frac{P_H}{P_L} \tag{1}$$

In what follows, we indicate the priority class by adding the subscript 'o' for the PCOUT class and 'i' for the PCIN class to S, P_H, P_L and other variables defined below. In the example of Fig. 2, $S_o = 10/6$, $S_i = 3/3$ for Agent1 and $S_i = 8/2$ for Agent2.

In an incoming allocation request, *FlexSigP* tries to distribute the BFs according to the different priorities and the corresponding ratios of resources (S_o and S_i). The process of allocating BFs to agents and to signatures for one agent is very similar: in both cases, the number of BFs needs to be readjusted in terms of the maximum number of resources available, the number of agents/signatures and their priorities.

The general expression for computing the maximum number of BFs for a low-priority agent/signature is

$$N_L(Q) = \left\lfloor \frac{Q}{\alpha_H \times S + \alpha_L} \right\rfloor \tag{2}$$

where Q is the maximum number of BFs to be distributed, and α_H, α_L are the total number of agents/signatures of high and low priority, respectively. The maximum number of BFs for a high-priority agent/signature is obtained from N_L as follows:

$$N_H(Q) = \lfloor N_L(Q) \times S \rfloor \tag{3}$$

with $Q = T$ when the calculation is for agents (T is the total number of BFs in the *FlexSigP* module). The allocation algorithm requires the computation of the corresponding values for agents of high and low priority, and for signatures of agents of high and low priority, resulting in the following values: N_{Ho}, N_{Lo} for agents, and N_{Hi}, N_{Li} for signatures of agents. For computing N_{Hi} and N_{Li} in expressions (2) and (3), Q should take the value N_{Ho} or N_{Lo} depending on the PCOUT priority of the agent. Therefore, a total of four values result: $N_{Li}(N_{Lo})$, $N_{Hi}(N_{Lo})$ for signatures of low PCOUT priority agents, and $N_{Li}(N_{Ho})$, $N_{Hi}(N_{Ho})$ for signatures of high PCOUT priority agents.

In the example of Fig. 2, in case a new incoming high-priority Sig3 request in Agent2, the new values for the maximum number of BFs for each signature, according to general expression would be $N_L(10) = \lfloor \frac{10}{2\times(4/1)+1} \rfloor = 1$ and $N_H(10) = \lfloor N_L(10) \times 4/1 \rfloor = 4$.

Restrictions. even at this high level, some restrictions should be applied in order to obtain a practical efficient hardware implementation. Namely:

(i) To simplify the allocation algorithm we consider the number of signatures per agent ($AS = \alpha_{Hi} + \alpha_{Li}$) a constant of the system. This restriction simplifies significantly the allocator (allow a single set of computations valid for all agents), while other parts of the system (hardware/software) can mitigate its effects. For instance if one agent needs more signatures, this can be done in several successive allocations.

(ii) The total number of agents $F = (\alpha_{Ho} + \alpha_{Lo})$ is bounded so that $F \times \mathrm{AS} \leq T$ (to have a lower bound of one BF per signature). This is solved by dimensioning the *FlexSigP* module according to the maximum number of threads supported.

(iii) S is also bounded to assure a minimum of one BF per signature. The bound for S_o is obtained from Equation (2), using $Q = T$ and with the condition $N_{Lo}(T) \geq AS$ (one BF for AS signatures for the low-priority agents). This results in the condition $S_o \leq 1 + [(T/AS) - F]/\alpha_{Ho}$. The bound for S_i is obtained also from Equation (2), using $Q = N_{Lo}$ (computed with the actual value

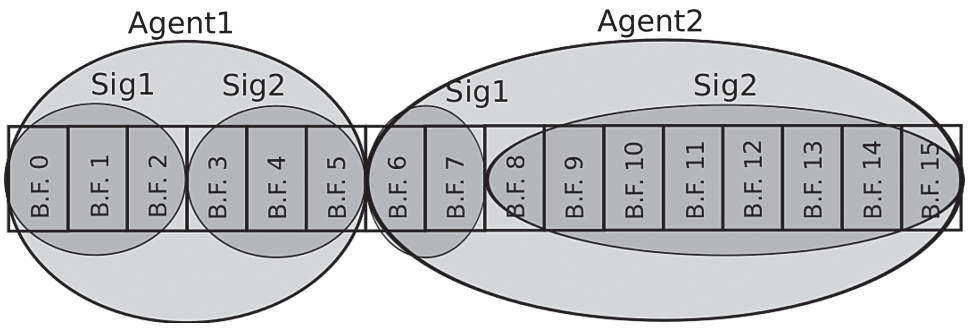

FIGURE 2. Example of PCOUT and PCIN. The PCOUT is regarding agents (low-priority Agent1 and high-priority Agent2) and the PCIN is regarding the signatures of each agent (Sig1 and Sig2 for each agent). In Agent1, both high and low priorities are the same, whereas in Agent2 Sig1 is low priority and Sig2 is high priority.

used for S_o) and with the condition $N_{Li}(N_{Lo}) \geq 1$ (at least one BF for the low-priority signature of the low-priority agents). This results in the condition $S_i \leq 1 + (N_{Lo} - AS)/\alpha_{Hi}$.

As in the symmetric case, a very accurate allocation algorithm leads to a very complex implementation that has a negligible improvement of performance compared with our implemented policy, which we explain below.

Algorithm 1 shows a high-level description of the allocation algorithm proposed. Our goal is to have a highly parallel implementation with distributed actions at the BF level (no centralized control).

For each signature, the order index of each BF is stored (N_x in BF number x of the *FlexSigP* module). The allocation starts with the computation of N_{Lo}, N_{Ho}, $N_{Li}(N_{Lo})$, $N_{Hi}(N_{Lo})$ (used for signatures of low PCOUT priority agents), and $N_{Li}(N_{Ho})$, $N_{Hi}(N_{Ho})$ (used for signatures of high PCOUT priority agents) as described before. Then the allocation is performed in a recursive process with two phases. The process is repeated for each signature for the requesting agent, starting from high-PCIN priority signatures. The idea is to repeat a process consisting in freeing resources and allocating them to each new signature. Specifically, in Phase I, BFs are freed so that each signature already allocated in the module keeps N_{Li} (in fact $N_{Li}(N_{Lo})$ or $N_{Li}(N_{Ho})$) or N_{Hi} (in fact $N_{Hi}(N_{Lo})$ or $N_{Hi}(N_{Ho})$) BFs depending on its PCIN and PCOUT priority (those BFs with Nx value—order index inside the signature—greater than N_{Li} or N_{Hi} are freed). In Phase II, the free BFs (those freed in Phase I and, for the first signature allocation, other BFs that were free because of previous deallocations) are allocated to the corresponding signature of the requesting agent, and new order index N_x values are generated for the BFs allocated to the signature.

The algorithm requires AS sequential steps, but the decisions to free or allocate are at the BF level and are done in parallel in all the BFs. This algorithm may favor the requesting agent, but this advantage is lost in the next allocation. Moreover, regarding the distribution of resources assigned to each agent, the policy favors the last low-priority signature allocated, assuring a lower bound of resources allocated to it of at least one BF. An 'ideal' policy would assign N_{Li} or N_{Hi} BFs to each signature, including those of the requesting agent, and some additional BFs (due to a nonzero remainder in the integer divisions) to some signatures (candidate signatures would be only those with the number of BFs greater than N_{Li} or N_{Hi} before the current allocation, and including the signatures of the requesting agent). However, the implementation of the 'ideal' policy is very complex, while we verified with our simulation framework that the results are very close to the actual implemented policy (within 2%).

Algorithm 1 *FlexSigP* allocation.

```
Nx: order index in a signature of BF x.
Compute N_Lo, N_Ho.
Compute N_Li (N_Li(N_Lo) and N_Li(N_Ho)).
Compute N_Hi (N_Hi(N_Lo) and N_Hi(N_Ho)).
for(i = 1 to AS) {
  /* For each signature of requesting agent */
  /* high PCIN priority first */
  Phase I:
    for each BF in the system {
      if low PCIN priority signature: free BF if (N_x > N_Li)
      if high PCIN priority signature: free BF if (N_x > N_Hi)
    }
  Phase II:
    allocate all free BFs to signature
    generate order index N_x for each allocated BF
}
```

4.2. Example: asymmetric algorithms applied to TM

It is illustrative to show how an specific application may take advantage of the asymmetric policies provided by the proposed scheme. Specifically, we use TM as a driver application for benchmarking *FlexSigP*. To scale in performance in the multicore/manycore era, applications need to be programmed for parallel processing. A parallel application running in a

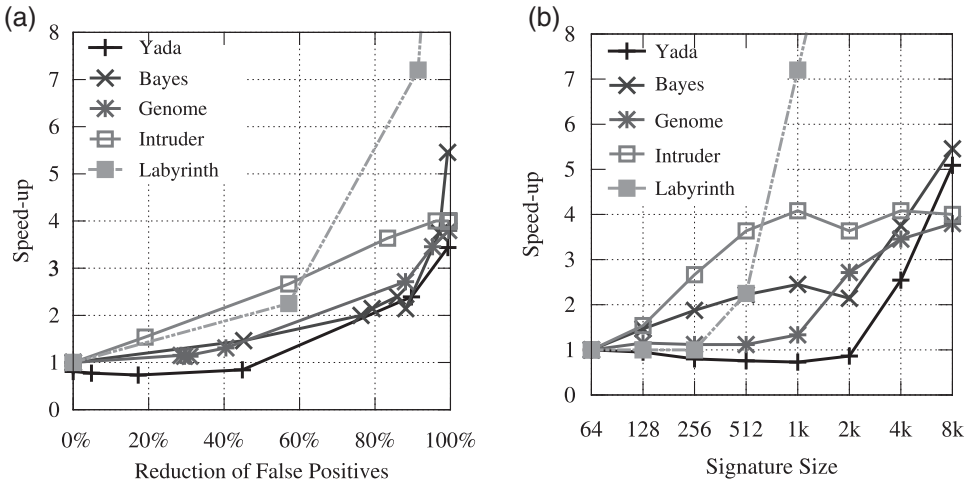

FIGURE 3. Speed up vs. reduction of false positives (**a**) and signature size (**b**).

multicore/manycore processor (several concurrent threads running in different cores) usually synchronizes through shared memory, so that specific mechanisms are necessary to do it reliably and efficiently. Locks are a traditional synchronization mechanism, but it is well known that they may lead to several problems such as deadlocks or priority inversion. Also, because locks use fine grain synchronization, programming and debugging are difficult, leading to low productivity and reliability issues [33, 34].

TM introduces coarse grain synchronization for easy programming and achieve performance with speculation and certain hardware support. Specifically, TM defines transactions as atomic sections of code that execute concurrently, in isolation and in a speculative way. TM can be implemented in software [35], but many proposals use hardware support [8, 27, 31] for performance reasons. Microprocessor vendors are already including certain hardware support for TM in their technology roadmaps [28–30], and further hardware support is expect in future generations.

There are several ways to take advantage of the characteristics of a TM system with a specific transactional code. Many TM systems use a read and a write signature to collect the read and the write sets of the transactions [12, 36]. Therefore, in a TM system the agents are the transactions which need to allocate two signatures ($AS = 2$), one for the read set, and one for the write set of the transaction. These signatures are used to detect atomicity violations of the transaction by other concurrent transactions. We may use each of the two levels of priority described in Section 4.1 to exploit the asymmetries between transactions and between the read and the write set signatures. The algorithm could be simplified by using one unified signature [12] for both read and write set, but we choose to use two conventional signatures to simulate a more complex scenario.

Benchmark programs show that there is a significant variability in the relative sizes of the read and write sets. In many occasions the read set is bigger than the write set, but sometimes they are similar, or even the write set is the bigger. The characteristics of the read and write set totally depend on the benchmark. Therefore, the false positive rate also depends on the characteristics of the read and write set. Thus, it would be of interest to adapt the sizes of the read or write signatures to optimize the overall false positive rate. To explore read/write asymmetry, the PCIN priority class is used.

On the other hand, benchmark programs have transactions with very different characteristics regarding the total size of the read/write sets and their access patterns. To take advantage of this, the allocation algorithm assigns resources based on the identifiers of the transactions using the PCOUT priority class. Thus, there is a set of more demanding transactions (that require bigger signatures to achieve a reasonable false positive rate), that are marked as high-priority transactions, and others that are more lightweight, and that are marked as low-priority transactions. The controller distributes the resources asymmetrically among transactions based on priorities, and may distribute resources with priorities between read and write signatures (using the PCIN priority class).

Our goal is to evaluate the reduction in the number of false positives by using *FlexSigP*. The number of false positives might be directly related with performance [37], since a false positive leads to an unnecessary conflict, and therefore one of the conflicting transactions has to stall or rollback and restart, with the consequent inefficiency. A certain reduction in the number of false positives implies a reduction in the penalties due to stall or rollback and restart. Of course, the overall effect on performance depends on the false positive rate in the application and the amount of transactional work performed.

To illustrate how the number of false positives affect performance, Fig. 3a shows the speedup for some instances of the STAMP benchmarks [38] (used for the evaluation of TM

systems) in terms of the reduction in the number of false positives. The data were collected from [39], where the reductions in the number of false positives were achieved by doubling the signature sizes from 64 bits up to 8K bits and simulating a hardware TM system [8]. Note that for some benchmarks even a 20% reduction from the base case, allows a significant speedup. Reductions of more than 80 % allows speedups of more than 2 for all the benchmarks considered, reaching speedups of more than 7 for some cases.

Another possibility in the context of TM is to share the *FlexSigP* module for different TM applications. Specifically, Fig. 3b shows the speedup variation with the signature size using the same source data as Fig. 3a. The behavior is highly dependent on the benchmark, leading to the possibility of improving results in a shared signature module by using priorities. For instance for a total of 9K bits and the benchmarks 'Yada' and 'Intruder' sharing the signature module, a symmetric policy would assign 4.5K bits to each. However, a better policy would be to assign 1K bit to 'Intruder' and 8K bits to 'Yada'. The priorities in *FlexSigP* can be used to achieve this kind of asymmetric distribution of resources. Note that in some situations, increasing the signature size reduces the speedup. This is due to some performance pathologies in TM applications, where some false positives may in fact prevent later more costly aborts.

Finally, to provide an intuitive view of the different possibilities in the context of TM applications, Fig. 4 shows a graphical example for a centralized signature module composed of $T = 16$ BFs, with a maximum of eight concurrent transactions, and for the case of three consecutive allocation requests. In each allocation, the agents allocate one read signature (marked with a 'r' in the figures) and one write signature (marked with a 'w'). We assume that BFs inside a signature are numbered from left to right in the module shown in the figures.

Figure 4a shows the statically fixed allocation of BFs, were one BF is allocated per signature. After the three agent allocations only 6 BFs are used of the 16 available. Figure 4b shows the case of an allocation in a *FlexSig* module. All the resources are used from the first request, and the resources are distributed with no priorities (the differences in the number of BFs are due to the integer arithmetic calculations). Figure 4c–e shows the examples of allocations with priorities in *FlexSigP* (the priorities in each case are indicated in the captions of the figures; see [40] for the details of the calculations performed).

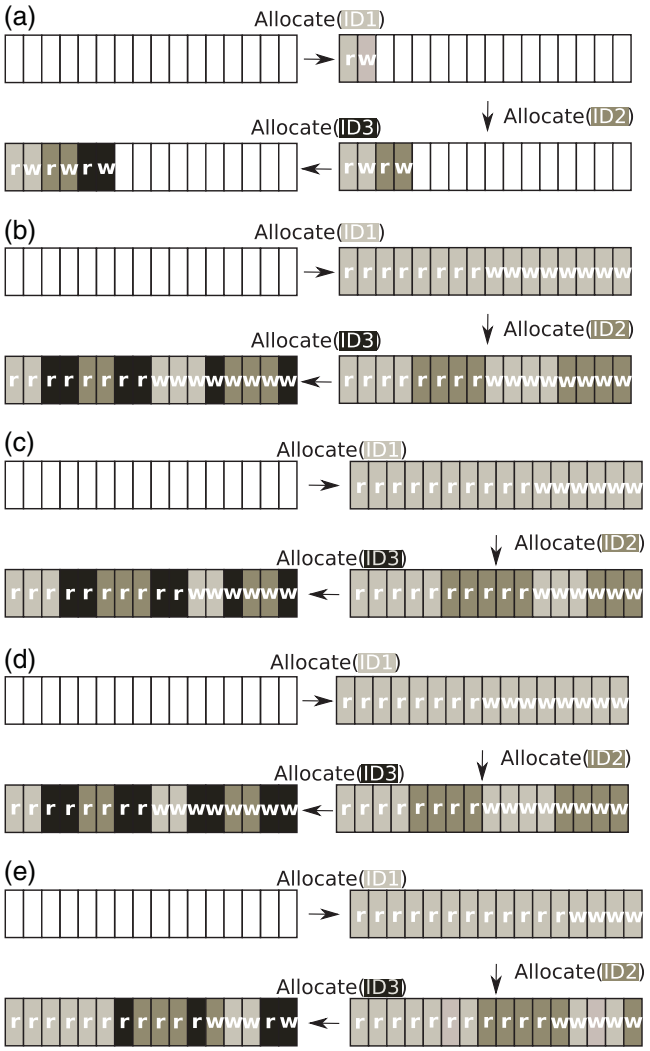

FIGURE 4. Examples of allocation algorithms. (**a**) Example of a statically fixed allocation of Bloom filters; (**b**) Example of the symmetric allocation algorithm in *FlexSig*; (**c**) Example of an asymmetric allocation algorithm with $S_i = 2$ (write signature of low-PCIN priority), $S_o = 1$ (no PCOUT); (**d**) Example of an asymmetric allocation algorithm with $S_o = 2$, $S_i = 1$ (no PCIN) and with transactions ID1 and ID2 of low priority, and ID3 of high priority); (**e**) Example of an asymmetric allocation algorithm with $S_o = 2$ and $S_i = 3$ (ID1 of high PCOUT priority, ID2 and ID3 of low PCOUT priority; write signature of low-PCIN priority).

4.3. Priority assignment

The optimal priority values change depending on the application, and to assign priorities to agents and signatures precisely is not trivial. There are several ways to deal with the problem.

The first way is to infer the priorities dynamically at runtime, adapting the priorities with an adaptive algorithm. The big problem of doing so is that we do not have enough information for adapting the priorities precisely: at runtime it is easy to know when a conflict is produced (in the TM context), but it is not possible to know if such a conflict is a real conflict or a false positive, and therefore it is not possible to adapt the priorities depending on the false positive rate.

An alternative would be to detect when a signature reaches a certain filling rate, but this would require a significant hardware overhead, without assuring good results, since the priorities would in fact depend on the ratio of filling rates of the

competing transactions, and the set of competing transactions may change during the application abruptly.

Another approximate alternative for long-term runs is to monitor the performance of the application with different parameters in the initial states of the execution, choose the best ones and maintain them for the whole execution of the program. For example, in a TM system, if a piece of code that contain transactions is repeated continuously during the whole execution of the program, in the initial states we could execute this peace of code with different parameters, and choose the best combination (based in the conflict rate). In our evaluation we choose our parameters following this philosophy: we try several parameters and we choose the best ones.

The second way to deal with the problem is statically, at compilation time. With static analysis we could estimate the average size of the transactions write and read set, and we could use this information to get a estimation of the priorities. However, there is information that is hardly inferable statically, such as the interference between transactions or even the size of the transaction (that may depend on the control flow). Moreover, based on our experience, the best priority assignment do not correspond always to the ratio of work set sizes.

Therefore, a practical, efficient and precise algorithm for assigning priorities probably would have to be a mix between static and dynamic techniques. Overall the problem requires a deep research to get a general solution, which is devoted for future work. On the other hand, for our specific implementation of priorities indicated by a three bit value, it required a manageable effort to detect good solutions by profiling the number of false positives in a set of runs.

5. HARDWARE IMPLEMENTATION FOR TM

This section describes the hardware implementation of the *FlexSigP* in the context of a TM system. We propose a hardware implementation for asymmetric allocation algorithms using read and write signatures. Despite the fact that this instance implementation is specifically designed for a TM system, it can be easily extended to other applications.

We assume a single transaction request to allocate the write and read signature. The hardware proposed implements the two classes of priorities described in Section 4.1. Symmetric allocations are performed by just setting the corresponding priorities S_i or S_o to one. Our implementation aims to provide a highly parallel low latency allocation operation in order to maximize the throughput of the accelerator.

We also assume that the priorities (P_{Hi}, P_{Li}, P_{Ho} and P_{Lo}) are known before execution. The calculation of the best match for each benchmark can be estimated by previous training executions, by static analysis (in the compiler) or a mix of both. The automatic static or dynamic calculation of the priorities is not straightforward, and it is a topic for future research.

5.1. Basic elements

Figure 5 shows the basic elements and signals of a superscalar *FlexSigP* module that can issue two instructions in parallel. The controller can be adapted to manage a greater number of parallel instructions at the expense of increasing complexity.

There are two requests managed by the controller in parallel (indicated as *request* in the figure), each one composed of the code of the instruction (**instr**), an address (*address*—used only in insert and check operations), the identifier of the transaction (*inID*), the identifier of the signature (*inSET*—used only in insert and check operations) and the allocation parameters (*alloc_param*—used only in allocation operations). The allocation parameters include P_{Hi}, P_{Li}, and the flags *pHSig* (equal to one for read signature of high priority), and *pOUT* (equal to one for a high-priority transaction).

The controller has several simple elements that are replicated for each BF to achieve maximum parallelism:

(i) **Bloom_filter$_x$**: it is a storage element composed of a register and a hash function.
(ii) **in$_x$**: input port for the address in the insert and check operations.
(iii) **out$_x$**: one bit output port for the result of the check operation.
(iv) **ID$_x$**: register to store the identifier of the transaction to which the *Bloom_filter$_x$* is allocated.
(v) **SET$_x$**: register of one bit to identify a read ($SET_x = 1$) or write ($SET_x = 0$) signature.
(vi) **Pi$_x$**: one bit register that indicates that a BF is part of a high-priority signature or a low-priority signature (PCIN priority).
(vii) **Po$_x$**: one bit register that indicates that the BF is part of a high-priority transaction or a low-priority transaction (PCOUT priority).
(viii) **N$_x$**: register that stores the order index of the BF in the signature. For each signature, all the BFs are numbered from one to the total number of BFs in the signature, in consecutive and ascendant order.
(ix) **control logic$_x$**: is a simple per BF control logic that performs the Check, Insert, Allocate and Deallocate operations by generating the required control signals (see Section 5.3).

There are also several signals in each BF that are activated by the **control logic$_x$** controller:

(i) **req$_x$**: this signal is used to control input and output multiplexers for each BF.
(ii) **clear$_x$**: this signal is used for the allocating and deallocating request to clear the corresponding BF.
(iii) **check$_x$**: is an enabling signal for reading the output of the BF.
(iv) **insert$_x$**: synchronous load signal for the BF.

Furthermore, the controller has an arithmetic calculations module that calculates the maximum number of BFs per signature

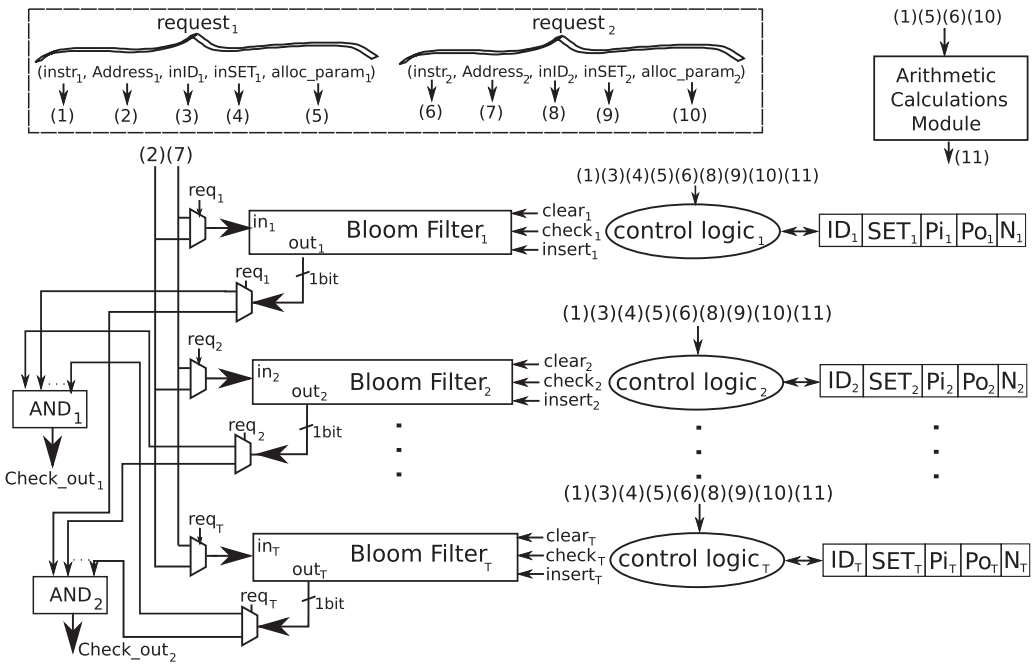

FIGURE 5. The basic *FlexSigP* elements in a two-way implementation (issue of up to two instructions).

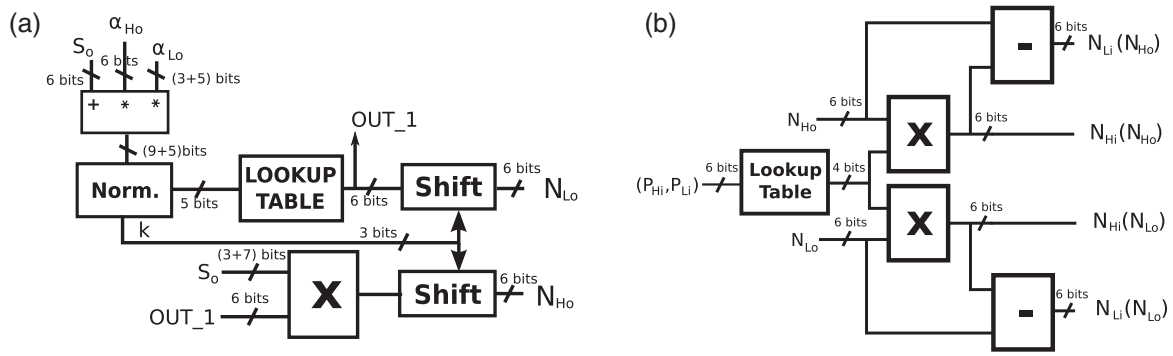

FIGURE 6. Arithmetic calculations for the allocation operation. (**a**) Calculation of the maximum number of BFs for high- and low-priority transactions and (**b**) Calculation of the maximum number of BFs per signature, depending on its priority and the priority of the transaction.

according to their priorities. These values are required by all the per BF control logic in each allocation request. Below, we describe this module in detail.

5.2. Allocation algorithm implementation: arithmetic calculations

In this subsection we show the implementation of the arithmetic part of the allocation algorithm. We present an implementation for specific values of some parameters in order to fully consider all possible optimizations. The extension to other values requires the redesign of some parts in order to optimize them for the specific parameters. Specifically, we present the design for $T = 64$ (total number of BFs in *FlexSigP*), and priorities defined by three bits (P_H and P_L are three bit numbers for both the PCIN and PCOUT priorities).

The calculations necessary to perform the allocation are described by Equations (2) and (3). In order to reduce hardware complexity, and since our allocation algorithm is in fact heuristic, we designed the arithmetic units so that an error of one unit is allowed with respect to the exact arithmetic implementation of these equations. These errors have a negligible effect in the final results (verified with our simulation framework), but allow a significant hardware reduction.

Figure 6a shows the calculation of N_{Lo} (Equation (2)) and N_{Ho} (Equation (3)). The divisor in Equation (2) is computed by a multiply–add operation. Since S_o is a constant during the execution of the application, we assume that this value is

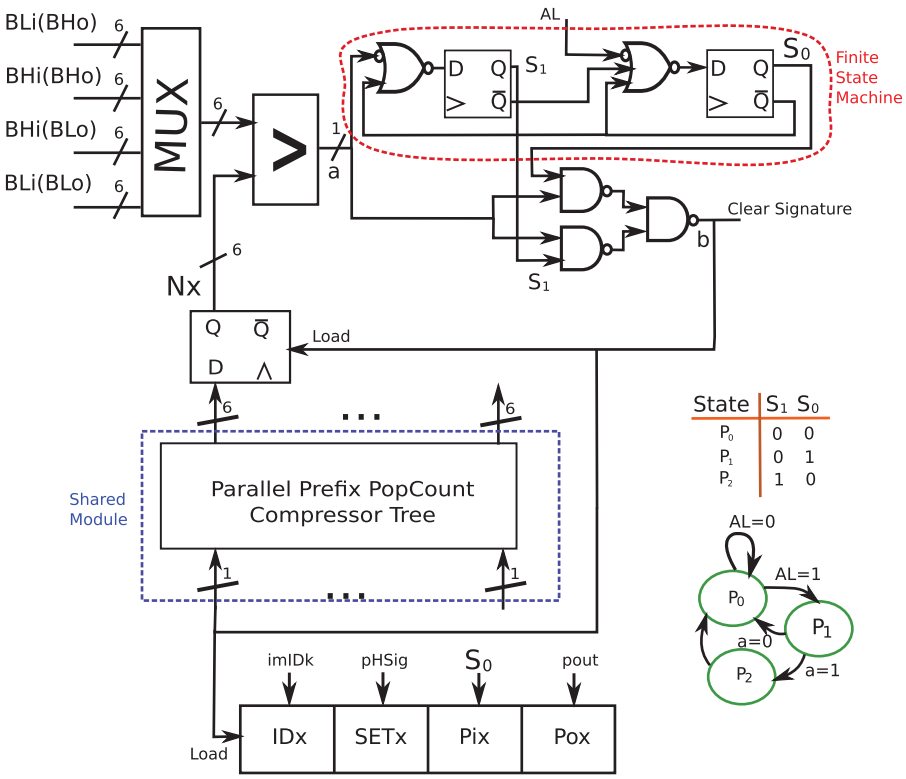

FIGURE 7. Control logic for each Bloom filter for allocations.

provided or computed elsewhere. At this point, according to Equation (2) a division operation is performed. The implementation of the division operation is simplified because: (i) the dividend is a constant of the system ($Q = T$); (ii) the output is a six bit integer number (the range is from 1 to T, with $T = 64$ in our implementation) and (iii) we allow an error of one unit with respect to the exact arithmetic calculation. This allowed us to use a look-up table to implement the division (due to the reduced number of bits of the input).

Figure 6a shows the number of bits (integer + fractional) required at the input/output of each module. As we show in the figure only the leading five bits of the denominator are necessary at the input of the divisor unit (look-up table), therefore the result of the previous adder is normalized and truncated to five bits. The normalization is compensated at the output with the shift operation. To calculate N_{Ho} (also in Fig. 6a), the output of the divisor (OUT_1 in the figure) is multiplied by S_o, and finally the result is shifted.

In Fig. 6b we show the calculation of the pair ($N_{Li}(N_{Lo})$, $N_{Hi}(N_{Lo})$) for low PCOUT priority signatures, and the pair ($N_{Li}(N_{Ho})$, $N_{Hi}(N_{Ho})$) for high PCOUT priority signatures. We rearranged the computation of Equations (2–3) to allow a more efficient computation due to: (i) $\alpha_{Hi} = \alpha_{Li} = 1$ (the number of signatures allocated is equal to two), (ii) S_i may have a different value in each allocation.

A look-up table is used to compute the result of $P_{Hi}/(P_{Hi} + P_{Li}) = S_i/(S_i + 1)$. The output of the look-up table is a number in the interval (0,1), with four fractional bits. This value is multiplied by N_{Ho} and N_{Lo} to obtain the maximum resources of the high-priority signature. The maximum resources of a low-priority signature are the remaining number of BFs, computed by means of a subtraction operation.

5.3. Control logic

In this section, we describe the control logic that generates local signals for each BF in Fig. 5. To simplify the presentation we show the more complex part, corresponding to the logic for controlling the allocation operations. Figure 7 shows the control logic in each BF to implement the allocation operation. It corresponds to a straightforward hardware implementation of the allocation algorithm shown in Algorithm 1.

A 4:1 multiplexer and a comparator implement the comparisons required in the Phase I of the loop body of the algorithm (signal a corresponds to the result of the comparison). The N_x value is stored in a six bit flip–flop, and indicates the order index of BF x in a signature. A very simple state machine controls the execution with inputs a (output of comparator) and AL (a signal that indicates the starting of an allocation operation by a transition from 0 to 1). The output of the state machine is

a single signal b that controls the actions corresponding to the Phase II of the loop body of the algorithm: a signal to clear the storage elements of the signature, a load signal for the registers that store state information ($\mathbf{ID_x}$, $\mathbf{SET_x}$, $\mathbf{Pi_x}$ and $\mathbf{Po_x}$), a load signal for the register that stores N_x and a signal used to recompute the order index in the signature.

To compute the N_x values (Phase II of the algorithm), a shared parallel prefix population count compressor tree is used. This unit computes the prefix addition of all the values of the b signal (0 or 1) of all BFs in the module. For instance for 8 BFs (BF_0 at the right and BF_7 at the left) and the sequence of b values 10 011 001, the result would be 43 332 111. The values computed are stored in the N_x register only for BFs with $b = 1$, resulting in $N_0 = 1$, $N_3 = 2$, $N_4 = 3$ and $N_7 = 4$.

The state machine has three states: P_0, P_1 and P_2. P_0 indicates that no allocation actions are performed, P_1 corresponds to the allocation actions for the high-PCIN priority signature and P_2 corresponds to the allocation actions of the low PCIN priority signature. The transition from P_0 to P_1 is triggered by a transition of the AL signal that indicates the starting of an allocation. The transition from $P1$ to $P2$ is done only for BFs with N_x values greater than the corresponding control value. The transition from $P1$ to $P0$ is performed in BFs with N_x values less or equal the corresponding control value (so no further allocation actions are performed on them). The transition from P_2 to P_0 is performed unconditionally after all the allocation actions are finished.

5.4. Hardware cost and latency

In this subsection, we use a high-level rough model to evaluate the hardware cost and latency of the proposed scheme. Although real implementations rely on optimizations done by synthesis tools on a specific standard cell library technology, this high-level analysis may give some insight about the added complexity of allocations with priorities. For the estimations we used the instance configuration sizes described in this section (64 BFs of 64 bits each).

We use a rough area-delay model based on logical effort [41]. This model is based on using cells with transistor sizing so that all the cells have the drive strength of the minimum size inverter. Buffering is introduced when necessary to optimize delays. We provide delays in FO4 units (1 FO4 is the delay of an inverter of minimum size with a load of four inverters), and hardware complexity in equivalent NAND-2 gate units. Interconnections loads are not taken into account. Optimizations such as gate sizing, low/high Vth etc. are not considered. Due to space limitations, we only present here the main results of the evaluation. Extended details can be found in [40].

Regarding hardware complexity, we obtained the following results:

(i) Cost of per BF allocator with priorities: 100 NAND-2
(ii) Cost of shared parallel prefix population count: 3340 NAND-2.
(iii) Cost of arithmetic calculations: 2235 NAND-2.

This leads to a total cost of 5600 NAND-2 for the shared modules, plus 100 NAND-2 per BF. To put these numbers in context, it is useful to estimate also the complexity of a single BF. We assumed an implementation based on a single port SRAM (with 6T cells), configured as a 8×8 array. The estimated complexity using our rough model is of about 450 NAND-2.

We conclude that for 64 bit BFs, the allocation with priorities requires about a 20% hardware overhead per BF. An upper bound of the overhead (considering only the hardware of the BFs and not the other supporting hardware) of the shared modules for a 64 BFs system is also ∼20%. Thus, the upper bound overhead in the evaluated system configuration is ∼40%, with respect to a centralized system with static allocation of BFs.

Regarding the overhead with respect to *FlexSig* using the same high throughput allocation scheme, we have to take into account that the arithmetic calculations part is simplified to just one look-up table, and that the per BF allocation hardware requires a 2:1 multiplexer instead of the 4:1 multiplexer. This results in a total overhead of ∼6% with respect to *FlexSig*.

However, in an actual implementation, this overhead is in fact highly dependent on the BF size and number of BFs in the system. Higher sizes and number of BFs reduce the overhead since the hardware complexity of the arithmetic calculations module and the parallel prefix population counter grows logarithmically with the number of BFs (the overhead per BF grows slightly, just one bit in the mux and the comparator for every doubling in the number of BFs). For instance, in a system with 64 BFs of 1024 bits each, an upper bound of overhead would be 3% with respect to a centralized module with static allocation.

Regarding latency, we obtained the following results:

(i) Delay of 80 FO4 for the arithmetic calculations.
(ii) Delay of 29 FO4 for the parallel prefix population count.
(iii) Delay of 40 FO4 for the per BF allocation scheme.

Regarding latency and throughput, the actual number of cycles of latency seen by the cores is determined by the core cycle time. For reference, we estimated a combinational delay of the cycle time of 23 FO4 for a current core implementation (see [40] for a discussion about this issue). Taking this value as a reference, this leads to a latency 4 core cycles for the arithmetic calculations, and 2 core cycles for the per BF hardware for allocation. Since the allocation requires two *FlexSigP* cycles with operations using the same hardware, this leads to a total of $4 + 2 + 2 = 8$ core cycles for an allocation. Regarding throughput, the arithmetic calculations can be simply pipelined by adding registers. This is not the case for the per BF hardware for allocation. Therefore, a new allocation can be performed every 4 core cycles.

TABLE 2. Configuration of the signatures used.

	Conventional	FlexSigP	Benchmarks
Cf.1	32 sigs × (256 bits, $k = 4$)	128 BF × 64 bits	intruder, vacation, yada, List, DList
Cf.2	32 sigs × (64 bits, $k = 4$)	128 BF × 16 bits	bayes, genome, kmeans, labyrinth, ssca2, Hash, Tree, TreeOver., Forest
Cf.3	–	128 BF × 8 bits	Eigen1, Eigen2
Cf.4	(a) 256 sigs × (64 bits, $k = 4$) (b) 128 sigs × (64 bits, $k = 4$) (c) 64 sigs × (64 bits, $k = 4$) (d) 32 sigs × (64 bits, $k = 4$)	(a) 1024 BF × 16 bits (b) 512 BF × 16 bits (c) 256 BF × 16 bits (d) 128 BF × 16 bits	Eigen3

6. SYSTEM ISSUES

Two implementation issues at the system level are: the software interface and the specific placement of the accelerator in the system. Regarding the software interface, specific ISA extensions should be provided to have a fully programmable accelerator. In order to support a wide range of applications with an efficient mapping to the accelerator, a careful design of the ISA extensions should be done.

There are several possibilities to place the module in the chip depending on the system, the target tool, the NOC (on-chip interconnection network) etc. In a general case, the module could be attached to the NOC in a similar way to other shared accelerators. This is similar to centralized BF bank module proposed in [10, 11] for TM applications. Depending on the size of the accelerator (number of BFs and size of each filter), a partitioned implementation could be considered, in a similar way to the bank organization in a cache system, to avoid the bottleneck of a centralized unit.

Furthermore, for dealing with high contention scenarios, our module could increase the number of read ports, in the same way that is done for the physical register file in OoO processors, or some high-performance caches, to allow several simultaneous reads to the same Bloom filters, at the cost of more hardware resources. Another complementary alternative is to use scheduling techniques [42] to mitigate contention, and aliviate the possible bottleneck in our shared module.

FlexSigP could be provided specifically to support novel important paradigms of parallel programming in multicore/manycore microprocessors (such as TM, data race detection etc). For this case, in a system with a ring NOC, the *FlexSigP* module could be placed at the cache controllers or directly connected to the NOC. In case of a system with a distributed directory cache coherence protocol, the module could be partitioned among directories based on address ranges, or the system could be clustered, by assigning one or more cores to a predefined directory. These two techniques could also be combined. As a specific illustrative example of use in this context, *FlexSigP* could be used as the conflict detection mechanism of a TM system similar to the IBM Blue Gene processor implementation [43] with minimum changes in the original system.

Furthermore, an alternative implementation could couple a *FlexSigP* module to each core (plus private L1 cache) to track the addresses of the threads that share the core. For instance, in the case of the IBM Power 8 processor [44], up to eight threads may share a single core. In this context, our module could be used per core by all threads of the core, just by adding a thread id per signature in the module.

7. EVALUATION FOR A TM SYSTEM

In this section, we evaluate the asymmetric policies in *FlexSigP* in the context of a TM system. Specifically, we use the information regarding the read and write sets from some well-known TM benchmarks for evaluating the reduction of false positives.

7.1. Experimental setup

We use RSTM [45] as TM system, and PIN [46] to instrument the transactional code (to gather read and write sets) and to simulate hardware signatures.

For testing our scheme, we chose some widely accepted TM benchmarks: STAMP [38] benchmarks, some micro benchmarks included in RSTM and Eigen benchmark [47] (a simple synthetic benchmark that can be configured to stress different TM characteristics).

Table 2 shows the four different configurations that we use to achieve the adequate false positive rates to compare the improvements in all the cases. Each configuration shows the parameters for the conventional signatures (statically allocated Bloom signatures), the parameters for *FlexSigP* and the benchmarks tested with that configuration. For example, Cf.1 uses 32 signatures (16 read signatures and 16 write signatures) of 256 bits and $k = 4$ (k is the number of hashes in the PBF) for conventional signatures, and a *FlexSigP* with 128 BFs of 64 bits (the same total number of bits as conventional signatures).

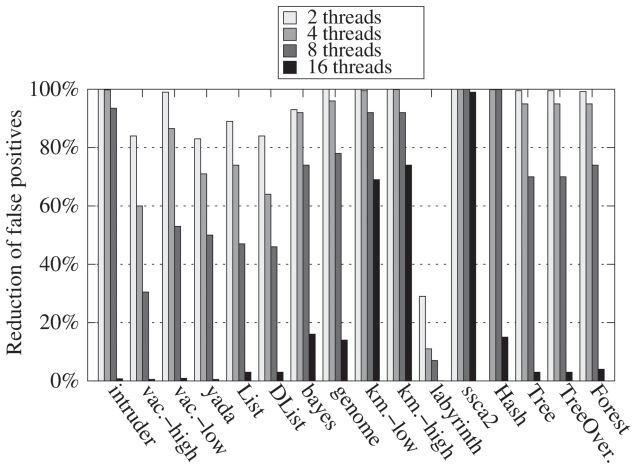

FIGURE 8. Reference graph that shows the percentage of decrease of false positives in symmetric *FlexSig* [26] compared with conventional signatures. The higher the bar, the better.

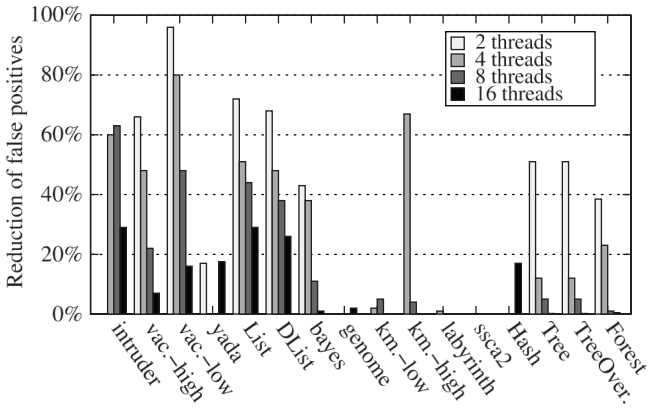

FIGURE 9. Percentage of decrease on false positives in *FlexSigP* [26] with PCIN priority class compared with symmetric *FlexSig*. The higher the bar, the better. The comparison of symmetric *FlexSig* with conventional signatures is in Fig. 8.

All the values of the priorities (P_{Hi} and P_{Li}), as well as the input of the benchmarks used in this evaluation are shown in Supplementary material [40].

7.2. Evaluation for the PCIN priority class

In this case, there is no priority for distribution of resources among transactions, but for each transaction a priority is provided to distribute the resources for the read and write set (PCIN priority class).

For testing this experiment, we chose STAMP [38] benchmarks and some micro benchmarks included in RSTM. We simulate the benchmarks using from 2 to 16 threads in a system with 32 conventional signatures (PBFs), and in a system with a *FlexSigP* module. In this experiment, we use the configurations Cf.1 and Cf.2 shown in Table 2. Cf.1 is used for the evaluation with the benchmarks that produce a higher rate of false positives. Cf.2 is used for the evaluation for benchmarks with lower levels of false positives.

For reference, Figure 8 shows the percentage of reduction on false positives of *FlexSig* (symmetric policies) with respect to conventional statically allocated BFs (higher is better, and a 100% reduction means no false positives). As we showed in [26], there is a significant reduction in the number of false positives (the reduction improves as the number of threads is reduced) due to the flexibility in resource management.

Note that the results of *FlexSig* compared with conventional signatures are worse when the number of threads increases. As we show in Section 7.4, this is not due to any scalability issue. The reason is because there are 32 conventional signatures in the system, which are all used only during the execution with 16 threads. Therefore, with less than 16 threads, there will be unused resources. Unlike conventional signatures, *FlexSig* always take advantage of all the resources of the module for any number of threads.

Figure 9 shows the percentage of reduction in the number of false positives in a *FlexSigP* implementing priorities for the PCIN class, compared with a *FlexSig* system implementing a symmetric policy. As shown in Fig. 9, for some benchmarks the reduction in the number of false positives is significant, achieving reductions higher than 50% in many benchmarks. Other benchmarks (such as 'ssca2', 'labyrinth' or 'genome') do not get a significant advantage from using the asymmetric policy, so they can be executed using a symmetric allocation algorithm. The results are usually worse when the number of threads increases because the average signature size decreases leading to more saturated signatures that reduce the advantage of the asymmetry.

7.3. Evaluation for combined PCOUT and PCIN priority classes

In this experiment, we explore the two priority classes (PCOUT and PCIN) using the Eigen benchmark. We use two types of transactions: the first type (ID1) is a transaction with a read set of 30 addresses, and a write set of 2 addresses, and the second type (ID2) has a read set of 4 addresses and a write set of one address. Half of the threads execute transactions ID1, and the other half execute ID2. We have two classes of asymmetry, one because of the different size of both types of transactions (PCOUT priority class), and the other because the read set is much bigger than the write set in both types of transaction (PCIN priority class).

The *FlexSigP* setup for this experiment is the Cf.3 in Table 2 (128 BFs of 8 bits). Furthermore, we use three priority configurations. The first configuration (PCIN) uses different priorities for the read and write signatures (as in the previous experiment), the second configuration (PCOUT) uses different priorities

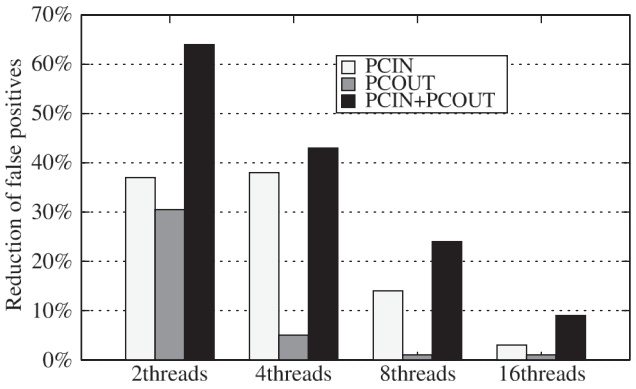

FIGURE 10. Percentage of decrease on false positives in *FlexSigP* implementing priorities for PCIN priority class, for PCOUT priority class, and combining both PCIN and PCOUT priority classes, compared with symmetric *FlexSigP* [26].

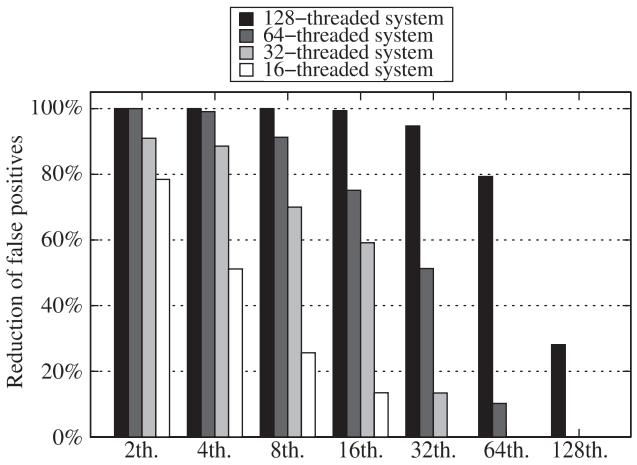

FIGURE 11. Percentage of decrease of false positives in *FlexSigP* (PCIN priority class) compared with conventional Bloom filters for Eigen benchmark with up to 128 threads.

for different transactions depending in their IDs, and the third configuration (PCIN + PCOUT) uses a combination of the two previous configurations.

Figure 10 shows the results of the three different configurations compared with *FlexSig*. It is clear that combining both kind of priorities the results are improved significantly. For instance, with two threads, the reduction in the number of false positives is under 40% when PCIN or PCOUT priorities are used alone. When both priorities are combined, the net effect is a reduction of ∼65%.

7.4. *FlexSigP* with a high number of threads

In this experiment, we show that *FlexSigP* can scale up to a high number of threads. The configuration Cf.4 of Table 2 is used to simulate an environment with a maximum number of 16, 32, 64 and 128 threads (configurations (a), (b), (c) and (d), respectively). We use the Eigen benchmark and *FlexSigP* implementing the PCIN priority class.

Figure 11 shows the reduction of false positives of *FlexSigP* (PCIN priority class) compared with conventional Bloom filters for a system with a maximum of 16, 32, 64 and 128 threads. *FlexSigP* achieves significant reductions, specially when the benchmarks are executed with less threads than the maximum allowed in the system. With this experiment we show that *FlexSigP* is getting worse when it runs with the maximum number of threads allowed in the system, but it has good scalability, as the behavior has the same pattern with systems up to 128 threads. The worst results are when a benchmark is executed with the maximum number of threads in the system. However, even in the worst case, *FlexSigP* clearly outperforms conventional Bloom filters.

8. RELATED WORK

Quislant *et al.* [48] have proposed a new reconfigurable asymmetric signature (ASYM) to deal with the asymmetry of the read and write sets in TM systems. The high-level idea is that the ASYM signature configures the number of BFs devoted to each data set. An ASYM signature is composed by k hash functions, each one associated with a register, and a mask register that provides a parameter (reconfigurable dynamically for each transaction) which establishes the sizes of the read and write signatures.

ASYM signatures comprise a more restricted scenario than our *FlexSigP*, as only manages asymmetry between read and write set in one transaction. Unlike ASYM, *FlexSigP* is a shared module, which deals with many transactions and more types of asymmetries, such as asymmetry among user transactions or asymmetry among applications. Also, *FlexSigP* is not intended to work only with TM, and it can be extended to work with other applications or tools. Because of its simplicity, ASYM does not require an allocation algorithm, and the size of the signatures would be constant from the beginning to the end of the transaction (in *FlexSig* the size can be reduced in a running transaction). In Section 8.1 we evaluate both, and the results show that *FlexSigP* achieves less false positives than ASYM signatures.

Korgaonkar *et al.* [49] propose to distribute the resources in *FlexSig* according to the size of the transactions. However, they do not propose any hardware implementation, nor do they describe the algorithm implemented in detail. In contrast, we propose a hardware approach of a general asymmetric algorithm (not only for TM) with several levels of asymmetry.

Scalable Bloom Filters [50], AdaptSig [9] or Dynamic Bloom Filters [51] propose alternatives in the same way: they expand signatures with more resources when the false positive rate reaches a prefixed level. For example, the Scalable Bloom Filters (SBF) are composed by one or more single Bloom filters; when the filters reach the fill ratio, a new filter is added to the SBF. Each successive BF is created with a tighter maximum error probability on a geometric progression, so that

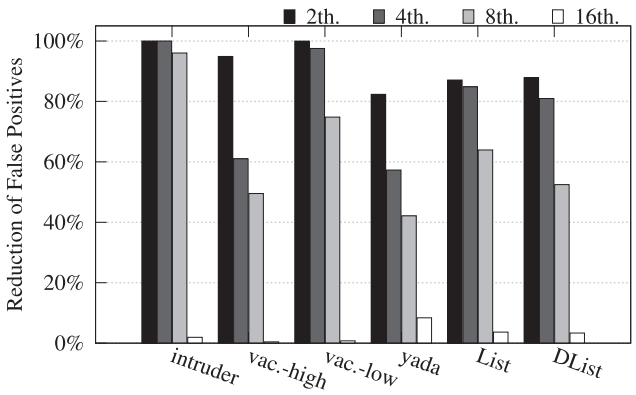

FIGURE 12. Percentage of decrease of false positives in *FlexSigP* implementing the PCIN priority class, compared with ASYM signatures.

the compounded probability over the whole series converges to some predefined value. With this method, the false positive rate is always bounded. However, it might be very difficult to implement these approaches in hardware, because an indefinite number of resources would be needed (it depends on the application).

The proposals in [10, 11] also use a centralized module of conventional signatures in a TM system. These schemes perform a static fixed allocation of signatures (resulting in signatures of fixed size), and they do not propose techniques to scale or to make them more flexible. Our *FlexSigP* could be used instead of the proposed modules in both proposals for reducing the number of false positives.

8.1. *FlexSigP* vs. ASYM signatures

We repeat the experiment of Section 7.2 for some benchmarks, but this time comparing the results of *FlexSigP* (with PCIN priority class) with the results of ASYM [48]. The system with ASYM signatures is configured with 16 ASYM signatures, each one composed by 8 BFs (each BF composed of a H3 hash and a 64-bit register), that accumulate the same storage capacity that our *FlexSigP* module. The detailed configuration is described in Supplementary material [40].

Figure 12 shows the results of comparing ASYM signatures with *FlexSigP* implementing the PCIN priority class. The bars indicate the reduction of false positives of *FlexSigP* over ASYM signatures. We can see that, for 2, 4 and 8 threads, the reduction of false positives in *FlexSigP* is very significant in comparison with ASYM signatures (up to 100%), because ASYM signatures do not take advantage of all resources when the number of threads is <16.

For this test, we only use a subset of the benchmarks that better represent the advantage of ASYM signatures over conventional Bloom filters. For the test used in Section 7.2 that are not shown in Fig. 12, the better results of ASYM are obtained with a symmetric configuration, and therefore the results for the ASYM signatures are the same as for the conventional Bloom filters shown in Section 7.2, that are also clearly outperformed by *FlexSigP*.

9. CONCLUSION

A *FlexSig* module might be an efficient accelerator for multiple applications in the multicore/manycore era. The module makes signatures flexible and adaptable to different situations and requirements. However, the algorithm for allocating signatures is very simplistic. In this work, we explored more involved techniques to allocate signatures with priorities. While the concept is simple, the trade-off between complexity of implementation and efficiency is not straightforward and required a careful exploration of the design space. We proposed *FlexSigP*, which implements two orthogonal levels of priorities with two possible states (high and low). We developed a high throughput hardware allocation algorithm (the critical operation) that is able to take advantage of priorities while keeping the hardware overhead at reasonable levels. Specifically we performed an extensive evaluation in the context of TM applications, achieving reductions of up to 60% in several real benchmarks when implementing the PCIN priority class compared with *FlexSig*. We also showed the advantages of combining both levels of priorities with a synthetic benchmark. We obtained rough estimates of the hardware overhead for supporting priorities, and the throughput of the system.

The asymmetric allocation policies proposed strengthen the case for *FlexSigP* as a flexible hardware accelerator to be used in a general purpose multicore/manycore microprocessor, extending the original concept of flexibility in *FlexSig* with the adaptation to application-dependent characteristics (asymmetry).

An interesting open problem for future research in *FlexSigP* is the automatic static or dynamic determination of the priorities by incorporating information from specific hardware counters.

FUNDING

This work was supported in part by Ministry of Education and Science of Spain, co-funded by the European Regional Development Fund (ERDF/FEDER), under contract TIN 2010-17541 and by Xunta de Galicia under contracts CN2012/151 and 2010/28. While performing most of this work, Lois Orosa was with the University of Santiago de Compostela.

SUPPLEMENTARY MATERIAL

Supplementary material is available at *www.comjnl.oxfordjournals.org*.

REFERENCES

[1] Bloom, B.H. (1970) Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, **13**, 422–426.

[2] Laufer, R.P., Velloso, P.B. and Duarte, O.C.M.B. (2011) A generalized bloom filter to secure distributed network applications. *Comput. Netw.*, **55**, 1804–1819.

[3] Donnet, B., Baynat, B. and Friedman, T. (2006) Retouched Bloom Filters: Allowing Networked Applications to Trade Off Selected False Positives Against False Negatives. *Proc. 2006 ACM CoNEXT Conf. Article No. 13*. ACM.

[4] Deng, F. and Rafiei, D. (2006) Approximately Detecting Duplicates for Streaming Data using Stable Bloom Filters. *Proc. 2006 ACM SIGMOD Int. Conf. Management of Data*, pp. 25–36. ACM.

[5] Fan, L., Cao, P., Almeida, J. and Broder, A.Z. (1998) Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol. *ACM SIGCOMM Computer Commun. Rev.*, pp. 254–265. ACM.

[6] Tarkoma, S., Rothenberg, C. and Lagerspetz, E. (2012) Theory and practice of bloom filters for distributed systems. *IEEE Commun. Surv. Tutor.*, **14**, 131–155.

[7] Wikipedia (2015) Bloom filter — wikipedia, the free encyclopedia. [Online].

[8] 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. *Proc. 2007 IEEE 13th Int. Symp. High Performance Computer Architecture, HPCA'07*, Washington, DC, USA, pp. 261–272. IEEE Computer Society.

[9] Peng, L., guo Xie, L., qiang Zhang, X. and yan Xie, X. (2010) Conflict Detection via Adaptive Signature for Software Transactional Memory. *2010 2nd Int. Conf. Computer Engineering and Technology (ICCET)*, April, pp. V2-306–V2-310.

[10] Casper, J., Oguntebi, T., Hong, S., Bronson, N.G., Kozyrakis, C. and Olukotun, K. (2011) Hardware acceleration of transactional memory on commodity systems. *SIGPLAN Not.*, **46**, 27–38.

[11] Ferri, C., Marongiu, A., Lipton, B., Bahar, R.I., Moreshet, T., Benini, L. and Herlihy, M. (2011) Soc-tm: Integrated hw/sw Support for Transactional Memory Programming on Embedded Mpsocs. *Proc. Seventh IEEE/ACM/IFIP Int. Conf. Hardware/Software Codesign and System Synthesis, CODES+ISSS'11*, New York, NY, USA, pp. 39–48. ACM.

[12] Choi, W. and Draper, J. (2013) Improving utilization of hardware signatures in transactional memory. *IEEE Trans. Parallel Distrib. Syst.*, **24**, 2230–2239.

[13] Qi, S., Otsuki, N., Nogueira, L.O., Muzahid, A. and Torrellas, J. (2012) *Pacman: Tolerating Asymmetric Data Races with Unintrusive Hardware*, pp. 1–12. Los Alamitos, CA, USA: IEEE Computer Society.

[14] Tuck, J., Ahn, W., Ceze, L. and Torrellas, J. (2008) Softsig: software-exposed hardware signatures for code analysis and optimization. *SIGPLAN Not.*, **43**, 145–156.

[15] Sung, H., Komuravelli, R. and Adve, S.V. (2013) Denovond: efficient hardware support for disciplined non-determinism. *SIGARCH Comput. Archit. News*, **41**, 13–26.

[16] Lin, P.-C., Lin, Y.-D., Lai, Y.-C., Zheng, Y.-J. and Lee, T.-H. (2009) Realizing a sub-linear time string-matching algorithm with a hardware accelerator using bloom filters. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, **17**, 1008–1020.

[17] Pontarelli, S. and Ottavi, M. (2013) Error detection and correction in content addressable memories by using bloom filters. *IEEE Trans. Comput.*, **62**, 1111–1126.

[18] Reviriego, P., Pontarelli, S., Maestro, J. and Ottavi, M. (2015) A synergetic use of bloom filters for error detection and correction. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, **23**, 584–587.

[19] Lin, Y.-D., Lin, P.-C., Lai, Y.-C. and Liu, T.-Y. (2009) Hardware software codesign for high-speed signature-based virus scanning. *IEEE Micro*, pp. 1–1.

[20] Lim, H., Lim, K., Lee, N. and Park, K.-H. (2014) On adding bloom filters to longest prefix matching algorithms. *IEEE Trans. Comput.*, **63**, 411–423.

[21] Tripathy, A., Ieong, K.C., Patra, A. and Mahapatra, R. (2013) A Reconfigurable Computing Architecture for Semantic Information Filtering. *2013 IEEE Int. Conf. Big Data*, pp. 212–218. IEEE.

[22] Chen, Y., Schmidt, B. and Maskell, D. (2013) Reconfigurable accelerator for the word-matching stage of blastn. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, **21**, 659–669.

[23] Liu, J., Jaiyen, B., Veras, R. and Mutlu, O. (2012) Raidr: Retention-aware Intelligent Dram Refresh. *Proc. 39th Annual Int. Symp. Computer Architecture, ISCA'12*, Washington, DC, USA, pp. 1–12. IEEE Computer Society,

[24] Lyons, M.J. (2013) Toward a hardware accelerated future. PhD Thesis, Harvard University.

[25] Horowitz, M. (2014) 1.1 Computing's Energy Problem (and what we can do about it). *2014 IEEE Int. Solid-State Circuits Conf. Digest of Technical Papers (ISSCC)*, pp. 10–14. IEEE.

[26] Orosa, L., Antelo, E. and Bruguera, J.D. (2012) FlexSig: implementing flexible hardware signatures. *ACM Trans. Archit. Code Optim.*, **8**, 30:1–30:20.

[27] Herlihy, M. and Moss, J.E.B. (1993) Transactional Memory: Architectural Support for Lock-free Data Structures. *Proc. 20th Annual Int. Symp. Computer Architecture, ISCA'93*, New York, NY, USA, pp. 289–300. ACM.

[28] Haring, R., Ohmacht, M., Fox, T., Gschwind, M., Satterfield, D., Sugavanam, K., Coteus, P., Heidelberger, P., Blumrich, M., Wisniewski, R., Gara, A., Chiu, G., Boyle, P., Chist, N. and Kim, C. (2012) The IBM blue gene/Q compute chip. *IEEE Micro*, **32**, 48–60.

[29] Advanced Micro Devices (2009) Advanced synchronization facility – proposed architectural specification Inc., 2.1 edition.

[30] Hammarlund, P., Martinez, A., Bajwa, A., Hill, D., Hallnor, E., Jiang, H., Dixon, M., Derr, M., Hunsaker, M., Kumar, R., Osborne, R., Rajwar, R., Singhal, R., D'Sa, R., Chappell, R., Kaushik, S., Chennupaty, S., Jourdan, S., Gunther, S., Piazza, T. and Burton, T. (2014) Haswell: the fourth-generation intel core processor. *IEEE Micro*, **34**, 6–20.

[31] Calciu, I., Gottschlich, J., Shpeisman, T., Pokam, G. and Herlihy, M. (2014) Invyswell: A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory. *Proc. 23rd Int. Conf. Parallel Architectures and Compilation*, pp. 187–200. ACM.

[32] Chang, F., Feng, W.-C. and Li, K. (2004) Approximate Caches for Packet Classification. *Twenty-third AnnualJoint Conf. IEEE Computer and Communications Societies, INFOCOM 2004*, pp. 2196–2207. IEEE.

[33] Cui, Y., Wang, Y., Chen, Y. and Shi, Y. (2013) Lock-contention-aware scheduler: a scalable and energy-efficient method for addressing scalability collapse on multicore systems. *ACM Trans. Archit. Code Optim.*, **9**, 44:1–44:25.

[34] Johnson, F.R., Stoica, R., Ailamaki, A. and Mowry, T.C. (2010) Decoupling Contention Management from Scheduling. *Proc. 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, ASPLOS XV*, New York, NY, USA, pp. 117–128. ACM.

[35] Shavit, N. and Touitou, D. (1995) Software Transactional Memory. *Proc. 14th Annual ACM Symp. Principles of Distributed Computing, PODC'95*, New York, NY, USA, pp. 204–213. ACM.

[36] Shriraman, A., Dwarkadas, S. and Scott, M.L. (2008) Flexible Decoupled Transactional Memory Support. *Proc. 35th Annual Int. Symp. Computer Architecture, ISCA'08*, pp. 139–150.

[37] Yen, L. (2009) Signatures in transactional memory systems. PhD Thesis.

[38] Cao Minh, C., Chung, J., Kozyrakis, and Olukotun, K. (2008) STAMP: Stanford Transactional Applications for Multi-processing. *IISWC'08: Proc. IEEE Int. Symp. Workload Characterization*, September.

[39] Quislant, R., Gutierrez, E., Plata, O. and Zapata, E. (2013) Ls-sig: locality-sensitive signatures for transactional memory. *IEEE Trans. Comput.*, **62**, 322–335.

[40] Orosa, L., Bruguera, J.D. and Antelo, E. (2016) Supplemental material to asymmetric allocation in a flexible signature module for multicore processors. www.comjnl.oxfordjournals.org.

[41] Vazquez, A. and Antelo, E. (2012) Area and Delay Evaluation Model for CMOS Circuits. Technical Report. http://www.ac.usc.es/node/1607.

[42] Cui, Y., Wang, Y., Chen, Y. and Shi, Y. (2014) Mitigating resource contention on multicore systems via scheduling. *Comput. J.*, **57**, 1178–1194.

[43] Wang, A., Gaudet, M., Wu, P., Amaral, J.N., Ohmacht, M., Barton, C., Silvera, R. and Michael, M. (2012) Evaluation of Blue gene/q Hardware Support for Transactional Memories. *Proc. 21st Int. Conf. Parallel Architectures and Compilation Techniques, PACT'12*, New York, NY, USA, pp. 127–136. ACM.

[44] Stuecheli, J. (2013) Next generation power microprocessor. *Hot Chips*.

[45] Marathe, V.J., Spear, M.F., Heriot, C., Acharya, A., Eisenstat, D., Scherer III, W.N. and Scott, M.L. (2006) Lowering the overhead of nonblocking software transactional memory. *Proc. 1st ACM SIGPLAN*, Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06).

[46] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J. and Hazelwood, K. (2005) Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, **40**, 190–200.

[47] Hong, S., Oguntebi, T., Casper, J., Bronson, N., Kozyrakis, C. and Olukotun, K. (2010) Eigenbench: A Simple Exploration Tool for Orthogonal tm Characteristics. *2010 IEEE Int. Symp. Workload Characterization (IISWC)*, pp. 1–11. IEEE.

[48] Quislant, R., Gutierrez, E., Plata, O. and Zapata, E. (2013) Hardware signature designs to deal with asymmetry in transactional data sets. *IEEE Trans. Parallel Distrib. Syst.*, **24**, 506–519.

[49] Korgaonkar, K., Garimella, K. and Veezhinathan, K. (2012) Size-proportional signature sharing for transactional memory systems. FASPP Workshop.

[50] Almeida, P., Baquero, C., Preguica, N. and Hutchison, D. (2007) Scalable Bloom filters. *Inf. Process. Lett.*, **101**, 255–261.

[51] Guo, D., Wu, J., Chen, H., Yuan, Y. and Luo, X. (2010) The dynamic bloom filters. *IEEE Trans. Knowl. Data Eng.*, **22**, 120–133.