IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received March 22, 2021, accepted April 26, 2021, date of publication May 4, 2021, date of current version May 14, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3077539

Migration in Hardware Transactional Memory on Asymmetric Multiprocessor

ZIVOJIN SUSTRAN AND JELICA PROTIC
School of Electrical Engineering, University of Belgrade, Belgrade 11120, Serbia

Corresponding author: Zivojin Sustran (zika@etf.bg.ac.rs)

This work was supported in part by the Serbian Ministry of Education, Science and Technological Development under Grant III44006(RZ62109).

ABSTRACT In this paper, a system is presented which implements transactions migration to an asymmetric multiprocessor in order to decrease the probability of conflicts and improve execution performance. Applications parallelization makes programming and testing much more difficult, so the goal is to avoid putting additional burden on a programmer. Therefore, the proposed solution should be fully implemented in hardware. In the asymmetric multiprocessor that is analyzed, all cores have the same instruction set, but they are asymmetric in terms of microarchitectural properties, so that $N - 1$ "small" cores are identical, while the Nth "big" core is different, as it provides better performance and higher capacities of its units. The idea is to perform transaction migration from the "small" core to the "big" one, based on the history of transaction execution. The experiments were performed using a significantly upgraded Gem5 simulator and eight parallel applications from the STAMP benchmark suite. The experimental results show the speedup and the rate of successfully executed transactions for five different multiprocessor configurations, including symmetric and asymmetric multiprocessors with or without transaction migration. The improvement our algorithm achieves for suitable applications is up to 14% (10% on average) in turnaround time compared to the solutions which do not make use of asymmetry for scheduling transactions.

INDEX TERMS Shared memory algorithms, multicore architectures, hardware transactional memory, asymmetric multiprocessor, thread migration.

I. INTRODUCTION

During the last decade, several commercial processors emerged addressing performance and ease of programming. In order to achieve more efficient use of a multi-core processor in terms of power consumption, asymmetric multi-core processors have been introduced. Such processors consist of multiple heterogeneous cores, so some of them are usually faster and the others are slower for certain purposes. Most of them have the cores on the same die which have identical instruction set (ISA). They differ in type (in-order or out-of-order), pipeline depth, issues width, etc. Those multiprocessors are called Single-ISA asymmetric multicore processors [1]. An application can be executed faster by mapping execution to the right kind of core [2]. Transactional memory [3] is a paradigm that eases programming and can achieve better performance. Creating a multithread program is easy because a programmer needs only to mark the critical section as a transaction. Transactional memory ensures that transactions are executed atomically with respect to other transactions. Transactions are running without waiting. If some problem occurs (like data race), then a transaction aborts and restarts. This approach is considered to be optimistic synchronization compared to pessimistic one, like the locked based. Furthermore, optimistic synchronization can improve performance if aborts occur rarely.

The prior research in Single-ISA asymmetric multiprocessors proposed moving a thread to the right core to improve performance. This move is achievable on a fine-grain level by moving only a small part of the thread [2], [4]–[6] or a coarse-grain by moving the whole thread [7]–[13]. Usually the fine-grain thread migration is performed by hardware, while coarse-grain is performed by the OS scheduler.

The published research on transactional memory is vast and covers various implementations which can be in hardware, in software, or in both. This research focuses on

The associate editor coordinating the review of this manuscript and approving it for publication was Christian Pilato.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

Hardware transactional memory (HTM) which is also implemented in commercial processors. Those implementations are best effort, which means that transactions are limited in size and if the programmer creates a large transaction, that transaction is impossible to execute, and the execution will fall back to conventional locking mechanism. Since each new generation of processors has increased number of cores, a problem may arise with scalability. If the number of threads is increased, the number of transactions executed in parallel can also increase, and this can lead to contention which will increase the number of aborts. The larger number of aborts means more wasted work and potential performance degradation. To alleviate this problem, scheduling techniques have been proposed in the previous research [14]–[21], which reduce parallelism when contention is high.

This work combines the above techniques to improve the performance of asymmetric multicore processors. Since the techniques are orthogonal, it is necessary to find an efficient way to obtain synergy in combining them. We try to improve performance of the applications by reducing contention. To the best of our knowledge none of the previous research has explored this problem in a similar way. However, some previous research can complement our own without any modification, since it only improves either asymmetric processors or transactional memory. Our solution is implemented fully in hardware and it is fully transparent to the programmer, i.e., the ISA is not changed, and the application code is not changed in any way, so legacy code should operate without modifications. The hardware modifications are small and require only several more kilobytes of memory on the die, which is easy to obtain by using modern technological process.

The main goals of our paper are:

- To introduce a novel algorithm and architecture for hardware-based migration of transactions on Single-ISA asymmetric multicore processors. The algorithm uses heuristics, which accounts for transaction length, success rate, and reasons for failure.
- To present the implementation of the transaction migration M-HTM system in the Gem5 simulator. The base transaction memory implementation is a replica of the most advanced commercially available solution, while the asymmetric multicore processor is a state of the art static (non-reconfigurable) solution.
- To show the evaluation of the presented algorithm on the STAMP set of benchmark tests for transactional memory. The improvement that our algorithm achieves of up to 14% (10% on average) in turnaround time compared to the solutions which do not take advantage of asymmetry for scheduling transactions.

The rest of this paper is organized in the following sections: Section 2 presents the background, motivating example, and the problem setting, followed by Section 3 which describes the architecture of the proposed system. Section 4 presents simulation analysis in detail, reports the experiments performed in this research, and presents their results. Section 5 discusses the related work. Section 6 concludes this paper with current problem remarks and future work.

II. BACKGROUND AND PROBLEM STATEMENT

A transaction is a sequence of instructions within a thread that must satisfy certain conditions. From the point of view of other threads, effects of the execution of one transaction must be as if all the instructions within that transaction performed simultaneously and indivisibly. In other words, atomicity of transactions must be provided for instructions reading from and writing to the memory, since these instructions make the effects of executing one thread visible to other threads. During the execution of the transaction, no other thread may access the data that the transaction reads or writes, otherwise the atomicity of the transaction is disrupted.

Let us assume a transaction X reads variable A, then another thread writes variable A, and finally at the end of the transaction X it writes the variable A. This sequence of instructions leads to an error called write after read. This error means that read and write to A performed by the transaction X are not executed indivisibly, and this kind of atomicity corruption is called a conflict. The conflict can occur between the transaction and any other thread, regardless of corrupting operations' execution inside of a transaction or inside the out-of-transaction code. A transaction always starts unconditionally regardless of the state of other processors/threads. During the transaction execution, it is checked whether the executions on other processors have disrupted the atomicity of the transaction. If a conflict is detected, the transaction is interrupted, the thread returns to the state before the transaction and execution of the transaction must be attempted from the beginning. In case the conflict occurred between two transactions, one of them must be terminated, so that the other transaction can continue execution, as the effects of the interrupted transaction are discarded. If the conflict happened between the transaction and out-of-transaction code, only the transaction can be interrupted. The effects of executing a transaction have to be discarded, and the effects of the operations that the transaction performed have to be reversible.

Hardware provides conflict detection and the reversal of transaction execution effects. To do so, the hardware must: 1) keep records of the data accessed within the transaction, 2) enable the exchange of information on shared data access between different processors and 3) make it possible to restore values that data had before the start of the transaction. The first and the second requirement can be provided in hardware for an arbitrary transaction length, e.g., the number of different data items that the transaction can access. For example, records of data that can be accessed can be performed using Bloom filter as in [22]–[24]. With the Bloom filter [25] it is possible to implement an unlimited set that is imprecise in hardware. In this case, imprecise means operation that checks that the value belongs to the set can return false positives. In this way, a conflict can be detected even though it does not really exist. This inaccuracy does not affect the logical correctness, but only the execution performance

of the application. Information about data modifications is exchanged in most implementations using the cache coherence protocol which in its essence allows the exchange of data access information between different processors. However, the third requirement depends on the length of transactions, respectively.

The hardware has to keep track which data were changed during the execution of the transaction, so that it could undo the changes in the case of conflict. This is usually done with a limited size buffer. The most common implementation in the proposed solutions and commercial processors is to keep the changed data in private processor caches, while preserving the old values in memory. Some of the earliest solutions are provided in [3]. In that case, during the execution of the transaction, removing of the data changed in the transaction from the cache must be prohibited, because otherwise that data would be made available to other processors, and the old value of data kept in memory would be lost. If the cache capacity is not large enough to keep all data accessed by a transaction, the transaction must be interrupted, because atomicity cannot be preserved. This is called a capacity overflow. Some of the proposed implementations try to overcome this limitation by some kind of virtualization, as described in [23], [26]–[29], but such a possibility does not yet exist in commercial processors due to the complexity of the solutions.

For some instructions it is not easy to undo the effects, such as system calls, access to some devices, etc. Transactions with these instructions cannot be performed successfully on commercially available processor. Some proposals to support the execution of such instructions in a transaction can be found in the open literature [30]. Bearing in mind that the implementation of such a solution is very complex, and the benefits of executing such instructions are relatively small, it can be assumed that these techniques will not be applied in commercially available processors in the near future. Therefore, developers are advised to write transactions so that they do not contain the aforementioned type of instructions [31].

The Single-ISA asymmetric processors usually consist of two types of cores. According to the research presented in [32], architectures with more than two types of cores do not bring performance benefits. However, there are architectures with many-type asymmetric cores [32]. The main goal of such solutions is to optimize power consumption rather than to achieve better performance, and power consumption is out of scope of this paper. Due to the differences between cores, some cores have better performance than others and they have higher capacities of their units. Since such cores require more transistors to be implemented, such cores will be called "big", while the rest will be called "small" cores. The "big" cores can be used to execute threads with high priority or computational intensity, while the "small" cores can execute threads with lower priority or less computational intensity [1], [33]. For example the "big" core can have superscalar pipelined processing in which instructions are executed outside the program sequence, multiple issues of instructions in the same cycle, and larger data caches,

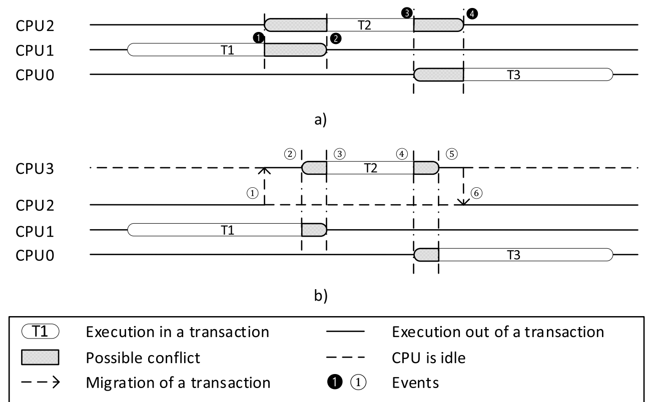

FIGURE 1. Execution of transactions on the a) symmetric b) asymmetric multiprocessor.

while the "small" core has a normal pipelined processing, where instructions are executed in the program order and have smaller data cache. Industrial implementation of such processors is already a commonplace and the commercially available products, by ARM and Samsung, are presented in [33], [34].

One thing to notice is that conflict can happen only during transaction execution. Since we have at our disposal a "big" core, we can use it to run a transaction and reduce its execution time. Our hypothesis is that the program, which uses transactional memory, could be accelerated on an asymmetric multiprocessor by accelerating transactions. Figure 1a) shows one execution of transactions on a symmetric multiprocessor. At the moment ❶ transaction T2 starts its execution. From that moment, transactions T1 and T2 are active, until transaction T1 is completed at time ❷. In that time period, if transactions T1 and T2 access the same data, a conflict can happen, that will cause the cancellation of one of the transactions. Thus, the operations of the canceled transaction are useless and must be repeated, thread execution lasts longer thereby reducing the speed of program execution. The same situation is from the moment ❸ to moment ❹ when transactions T2 and T3 are active at the same time.

The longer the time periods in which at least two transactions are active, the greater the probability of conflicts. Our assumption is that the length of such time periods can be reduced if the same program is executed on an asymmetric multiprocessor. The basic idea is to perform one of the transactions on a "big" core. In that way, the duration of that transaction execution decreases, so the time periods in which at least two transactions are active become shorter, which means that the likelihood of a conflict between two transactions also decreases. There are several solutions to perform one of the transactions on a "big" core, which will be considered later in this paper. In this example, one of the transactions is moved to execute on the "big" core, just before it starts the execution. Figure 1b) shows an example of executing the same transactions as in Figure 1a), but on an asymmetric multiprocessor with transaction migration. The given example is idealized, i.e. it shows the best case, because

it is difficult to detect how transactions overlap during the execution and it is assumed that both migration and execution of the transaction on ''big'' core take less or equal time as execution on slower core. At time ①, when the CPU2 core should start executing transaction T2, the CPU2 core migrates the execution of the transaction to the ''big'' CPU3 core. By the time ②, migration is taking place, which consists of stopping the execution on the CPU2 core, transferring the context to the CPU3 core, and starting execution. No conflict can occur during that time because the transaction T2 is not active. It is activated only at time ②. From that moment to the moment ③, there are two active transactions T1 and T2.

This period is shorter than the execution shown in Figure 1a) because the transaction T2 started later, and thus the probability of conflict became lower. From moment ④ to moment ⑤ transactions T2 and T3 are active. This period is also shorter than the period from ❸ to ❹ because transaction T2 executes on the ''big'' CPU3 core, so it ends earlier. From the moment T2 finishes its execution, the transaction returns to CPU2 by the moment ⑥. There is no difference in execution speed in the two presented cases if there is no conflict and cancellation of transactions; however, if there is a conflict, execution will slow down. If the scenario shown in Figure 1b) occurs often, the probability of a slowdown due to transaction cancellations will be smaller, which can result in better program execution performance on an asymmetric multiprocessor.

This example shows that the transaction in the thread running on a CPU2 core executes in a time interval of similar duration in both cases (including the time to migrate). This is a simplified presentation which neglects some details in order to explain the basic idea in an easy manner. In the actual execution, the two times will be different, depending on the time it takes for the transaction to migrate, as well as the speedup in the T2 transaction execution on the ''big'' core. The time spent on transactions migration is clearly an overhead cost, which can lead to a slowdown in thread execution. To justify the transaction migration cost, this overhead cost must be less than the cost of useless work performed by the transaction before cancelation.

In addition to the described potential benefits, execution on a ''big'' core can make it possible for some transactions to be performed at all. A ''big'' core may be faster because it has larger buffers and more cache memory, so that the core has a larger capacity to store the speculative data of a transaction. Higher capacity allows for some transactions to be executed, although they cannot be performed on a ''small'' core due to transaction cancellation caused by capacity overflow. These benefits could apply only on specific applications, whose sets of speculative data in transactions are big enough that they cannot be executed on a ''small'' core, and they can be executed on a large one.

There are several ways to ensure transactions execution on a ''big'' core. A trivial way is to run one thread of the program on a ''big'' core and make all its transactions execute faster than if the thread were executed on a slower core. This is easy to implement and speeds up all transactions in a single thread and those transactions maybe will not be in conflict with other execution. In this way it becomes unlikely to have a scenario like the one presented in Figure 1. Some transactions may already have small probability of conflict (short transactions or those with a small data set), so acceleration of such transactions will not bring any benefits. This solution is static because it does not allow reduction of conflicts between transactions within the threads that are executed on slower cores.

Another solution is that the operating system controls whether a transaction should be executed on a ''big'' core. One simple approach is to have the operating system spread the threads across the available cores, so that the thread with the maximal prediction of conflicts executes on a ''big'' core. For such an approach it is necessary to define an algorithm to predict which thread will have the most conflicts transactions and implement it in the operating system scheduler. In order for the software to perform that prediction, it is necessary to provide access to the information on executed transactions (both successfully executed and canceled). In most commercially available transactional memory implementations, e.g., [31], it is only possible to retrieve information about the successful execution of the transaction and the reason for cancellation in case of failure. For some other information type it is necessary to expand the instruction set of the processor. A potential problem with this approach is that a transaction migration can be carried out in a relatively distinct moments in time, but between these moments a large number of transactions can be executed (those that should be executed on a ''big'' core and those that should not). Such a solution would be suitable for those programs that have execution phases in which transactions of one thread have frequent conflicts with transactions from other threads. Another problem that can occur is the frequent change of the core on which the thread is executed. This can lead to a ''cold start'' problem, where a thread which starts running on another core does not have data in the cache that it needs. This will cause a lot of cache misses that can significantly slow down thread execution. A more complex approach would be that at the beginning of each transaction the software decides whether the transaction should be executed on a ''big'' core. If it should, hardware performs the transaction migration to the ''big'' core. This approach requires further change in the instruction set, namely, to add appropriate instructions which start the transaction with migration requirement. Execution of the prediction algorithm by the operating system can compromise execution performance if it happens to be on a critical path.

The third way is to implement an algorithm in the hardware in order to predict whether the transaction should be migrated to a ''big'' core. Since the transaction is a clearly defined so that it begins with an instruction to start the transaction and ends with either an instruction to end the transaction or a transaction cancellation, the hardware can easily detect at which point execution needs to be migrated

from one core to another and it is not necessary to change the instruction set and the software. It is necessary to keep records of transactions so that the algorithm can determine whether a transaction should be migrated to a "big" core. All the information can be accessed by the algorithm without the need to change the instruction set. Since the algorithm is implemented in hardware, its execution can be done in parallel with the program execution. This execution of the algorithm will not affect the performance of the program execution. Since the algorithm is implemented in hardware, it must be simple enough for technical implementation, considering its complexity and consequently the necessary space on the chip. Software solutions are in advantage because they can implement significantly more complex algorithms. Another advantage of software solution is that algorithms can be significantly configurable, to become fine-tuned for each type of program, while configurability of hardware solutions can be reduced to the change of some integer parameters. It has been considered in the paper that the most serious problems in the new solution application refer to changing the instruction set and performing the necessarily severe changes of the already existing programs, which may slow down the penetration of some techniques into commercial products. For that reason, a hardware solution that can be minimally configured has been considered in this paper.

Both the hardware and software solution with the migration of threads on level of individual transactions suffer from the same problem, which can make such solutions unusable. In thread migration, we need to stop the pipeline processing on the core, pack the context of a thread, copy the context to another core, and run the pipeline processing on that core. If on that other core some other thread was executed in another address space, some parts of the cache must be invalidated, as well as buffers for virtual memory mapping, etc. The entire process of that change can take several million cycles at the worst case scenario [35]. That time is too long for frequent migration to be beneficial, so any solution that uses migration of threads often would not improve performance of the execution. However, the characteristics of transactions are such that some things that would make the time of migration long are prohibited in HTMs. For example, the whole context does not have to be copied, but only the part that is saved on starting the transaction. Migration will only be performed if on a "big" core the thread was previously performed in the same address space as the thread the transaction of which should be migrated. In this way, the migration can be achieved in several orders of magnitude less time, which allows implementation of a solution that will improve execution performance.

Context transfer during migration can be carried out in two ways. The first way is to keep the whole context on the thread stack and to pass only the address of the next instruction and the address of the top of the stack to the target core on the occasion of migration. Then, on the target core the entire thread context should be taken from the stack before the execution of the transaction begins. The same procedure occurs when returning execution to the original core if the transaction was successful. In this way, the context transfer is done through a cache coherence protocol. Another way is to pass the entire context along with the transaction migration message. A mechanism for preserving the current content of architectural registers can be used in this case [36], which also exists due to the implementation of transactional memory. This content needs to be sent with the proper messages to the other core which should continue execution. From the point of view of performance and complexity of implementation, there is no obvious advantage of one or another solution. It is necessary to investigate which solution is more suitable for each microarchitecture. In case of unsuccessful migration of the transaction, both solutions can use the already existing transactional memory storage mechanism and do not need to restore the context at the original core.

An important design decision is what a "big" core would do until it executes the migrated transaction. The simplest solution is to do nothing. In this way, as soon as a request for execution of migrated transactions occurs, it can be accepted. Another approach is that a "big" core executes the program thread like the other cores. When a request to execute a migrated transaction arrives, it should stop executing its thread in order to execute the migrated transaction. Correctness of execution must be preserved, so the pausing of the thread must be performed outside of any transaction. This approach increases the potential for parallelism but slows down the migration itself. Nevertheless, it is not possible to claim in advance which approach is the better one. In the following chapters, both approaches will be analyzed.

III. M-HTM DESIGN

The algorithm for transaction migration, which we call M-HTM, is fully implemented in hardware. There is no need to change the application code in any way for its proper working. It is assumed that the processor supports transactional memory and has explicit instructions for starting and ending the transaction. For all other details the specific implementation of the transactional memory is not crucial, and the algorithm can be adapted to the implementation details of the transactional memory (in most cases there is no need to change anything). The multiprocessor must be asymmetric. All cores have the same instruction set and are asymmetric in terms of microarchitectural properties.

The proposed algorithm is designed for such a set of cores that if there are N cores, $N - 1$ cores are identical and considered "small" cores, while the N-th core has different microarchitectural properties and considered a "big" core. The existence of more than one "big" core is possible. The proposed algorithm can be modified in order to support processor architectures with multiple "big" cores, but it goes beyond this research and may be the topic of future work.

A. ALGORITHM

Based on the transaction execution history the algorithm decides whether this particular transaction should migrate to a "big" core. Moreover, for each transaction, records are

		SMALL SUCCESS		
		Y/NA	N	
			SMALL FAIL TYPE	
BIG SUCCESS	BIG FAIL TYPE		C/NA	O
N	O	No	No	No
N	C/NA	No	?	Yes
Y/NA		No	Yes	Yes

Y – successful
N – not successful
C – conflict abort
O – capacity overflow
NA – not available
Yes - migrate
No – do not migrate
? – test migration

FIGURE 2. Actions in particular cases.

kept referring to its success, the failure type and the transaction size as well. Information about success and failure type are kept separately for execution on the "small" and the "big" core. Records of success keep track of the information whether the transaction has been completed successfully in the earlier few executions or it was canceled. Records of failures keep track of whether the transaction has been aborted in the earlier few failures because there was a conflict with another transaction or because of a capacity overflow. The transaction size records keep track of whether the transaction is long enough to justify the migration. The dynamic transaction length, expressed in the number of instructions, must be made available to the algorithm. In order to keep that record each core, while executing a transaction, counts each executed instruction. The counter is reset when the transaction starts. The counter value is passed to the algorithm after the transaction is completed. Dynamic transaction length can vary significantly, due to the dynamics of program execution or because the transaction is aborted by conflict at any time. The algorithm, therefore, determines whether the transaction lasts long enough for migration based on several earlier executions.

The current state of the history is determined based on data in the records. It can be determined whether the transaction should be migrated or not, based on the current state of history. Figure 2 shows which actions are required depending on the state of history for an individual transaction. If the transaction successfully executes on the "small" core, there is no need for migration, because no conflicts occur, so there is no need to reduce the likelihood of their occurrence. Also, there is no need to migrate the transaction in the case that the transaction does not execute on the "big" core due to capacity overflow. In that case, the best solution is to continue with the execution using locking mechanism, because the transaction cannot be executed on any core of that multiprocessor. The transaction should be migrated in the case when it can execute successfully on the "big" core and does not execute on the "small" core, whatever the reason may be. In the case the transaction cannot be performed on a "small" core due to capacity overflow, it is desirable migrate it to a "big" core, if there is a chance that it can be successfully executed on it.

If the transaction was canceled while executing on both the "small" and "big" cores, the state history does not help to determine whether a transaction should be migrated (state with the "?" on the Figure 2). In that case, there is elevated level of parallelism and it causes frequent conflicts between transactions. Such an execution phase must eventually end with the execution using locking mechanism, that will provide the necessary serialization and reduce parallelism. In that case, it is better to avoid wasting time on the transaction migration by ensuring the transfer to the execution utilizing the locking mechanism on the "small" cores. The duration of such a state is not easily predictable. While such a situation lasts, the migration of the transaction is periodically allowed with the aim of reacting to the change as soon as possible. If the migration is successful, transition is made to a state in which the transaction is successfully executed on the "big" core and the transaction is not migrated further.

The overall algorithm pseudo code is presented in Listing 1. Algorithm time complexity is constant since it does not require any looping through elements. The spatial complexity is also constant since the algorithm requires five counters and four flip-flops to maintain state. Several adders and comparators are required to implement the algorithm. Since width of operands is expected to be small (four in our experiments), circuit critical path is expected to be low and the algorithm can execute at most in two clock cycles on high frequency rate in modern processors.

The number of transactions that can be migrated depends on the configuration of the "big" core. That number is equal to the number of threads that the core can execute in parallel. If the number of "small" cores that need to migrate their executions is greater than the maximum number of migrated threads, it is possible to cancel migration or wait for some of the migrated threads to finish. It is preferable to wait if the expected waiting time is shorter than the time that would be spent on useless cancelations and repeated executions of a transaction. The total number of transactions that have been migrated and are awaiting migration should be less than the number of "small" cores, because otherwise complete serialization of transactions could occur, which can

```
Algorithm: Decision on migration
Input: transaction's result, failure type, and length, core type
which executed transaction
Output: decision whether to migrate or not

procedure hysteresis(counter, state, value)
  counter = counter + value
  if (state is NO and counter >= UPPER_LIMIT)
    state = YES
  else if (state is YES and counter <= LOWER_LIMIT)
    state = NO
    end if
  end if
end
function to_migrate(core, result, failure_type, length)
  State: small_scnt, small_success = YES, big_scnt,
  big_success = YES, small_ccnt, small_conflict = YES,
  big_ccnt, big_conflict = YES, max_length = 0
  success_value = 1
  if (result is FAIL)
    success_value = -1
  end if
  failure_value = 1
  if (failure_type is OVERFLOW)
    failure_value = -1
  end if
  if (core is BIG)
    hysteresis(big_scnt, big_success, success_value)
    hysteresis(big_ccnt, big_conflict, failure_value)
  else
    hysteresis(small_scnt, small_success, success_value)
    hysteresis(small_ccnt, small_conflict, failure_value)
  end if
  if (max_length < length)
    max_length = length
  end if
  if (max_lenght < LENGTH_LIMIT or small_succes is YES)
    return NO
  end if
  if (big_success is YES
    or (small_conflict is NO and big_conflict is YES))
    return YES
  end if
  return NO
end
```

LISTING 1. Algorithm which decides whether to migrate transaction, based on history of execution.

significantly reduce program execution performance. In our algorithm, it is ensured that a small number of transactions can be on hold (number can be configurable on application basis, but limited with the size of the waiting queue in ''big'' core), so the chance for migration is provided, while the parallelism is not significantly reduced. If the ''small'' core tries to migrate the transaction, and there is not enough space to wait, the ''small'' core gives up the migration and tries to execute the transaction. It should be noticed that due to this limitation, the state in which the transaction should be migrated to the ''big'' core is not permanent, since it is possible to attempt to execute a transaction on a ''small'' core. If these attempts are often successful, it is possible to go to the state in which transactions on a ''small'' core execute successfully and the algorithm can block the migration of that transaction.

In the case when there is no history of the transaction execution (for example, the transaction is executed for the first time), it is assumed that the transaction can be successfully executed on both core types, just like in the case that there was no cancellation of the transaction. It is a state of history in which there is no migration, so the preference is given to the execution of transactions on ''small'' cores. Each time a transaction finishes its execution successfully, the record of execution on the ''big'' core is invalidated, as well as record of the failure type (i.e. the state is like the transaction was never executed on the ''big'' core). In this way, a non-intuitive transition between states is blocked. One example is that a transaction in some executions was too long, so it caused capacity overflow. After some time, the length of the transaction is reduced, so it begins to execute successfully on ''small'' cores. If the transaction starts unsuccessful executions caused by the conflict, it may happen that the state of history is such that the transaction does not migrate because it has been unsuccessfully executed a long time ago on the ''big'' core due to capacity overflows.

Note that the state of history is determined based on global information about the transaction, i.e. information about the transaction that was executed on any core is used. The transactions that execute over the same piece of code are considered identical. In addition to global information the ''small'' core also considers local information. If the earlier execution of the transaction on that core was successful, then the next time it is executed without migrating. Also, if the ''small'' core migrated the transaction to the ''big'' core and it was canceled there, the ''small'' core next time executes that transaction without migrating. In this way, the immediate local history takes precedence in relation to the global history. Therefore, we avoid a large number of migrations, those with a small chance to be useful.

B. IMPLEMENTATION

For the algorithm to work properly, it is necessary to collect information about the executed transactions that should be stored by hardware. Figure 3 shows an asymmetric multiprocessor with added information units designed to keep data about transactions. The storage for information about transactions is centralized, and one possible place for its placement is the small memory inside the ''big'' core. This small memory will be called the central register (CR). Whenever the ''small'' core executes a transaction it sends information about that transaction to the ''big'' core, which updates the state of the CR. The ''small'' core should check the CR before starting the transaction, in order to find out whether the transaction history is such that the migration should be performed and in

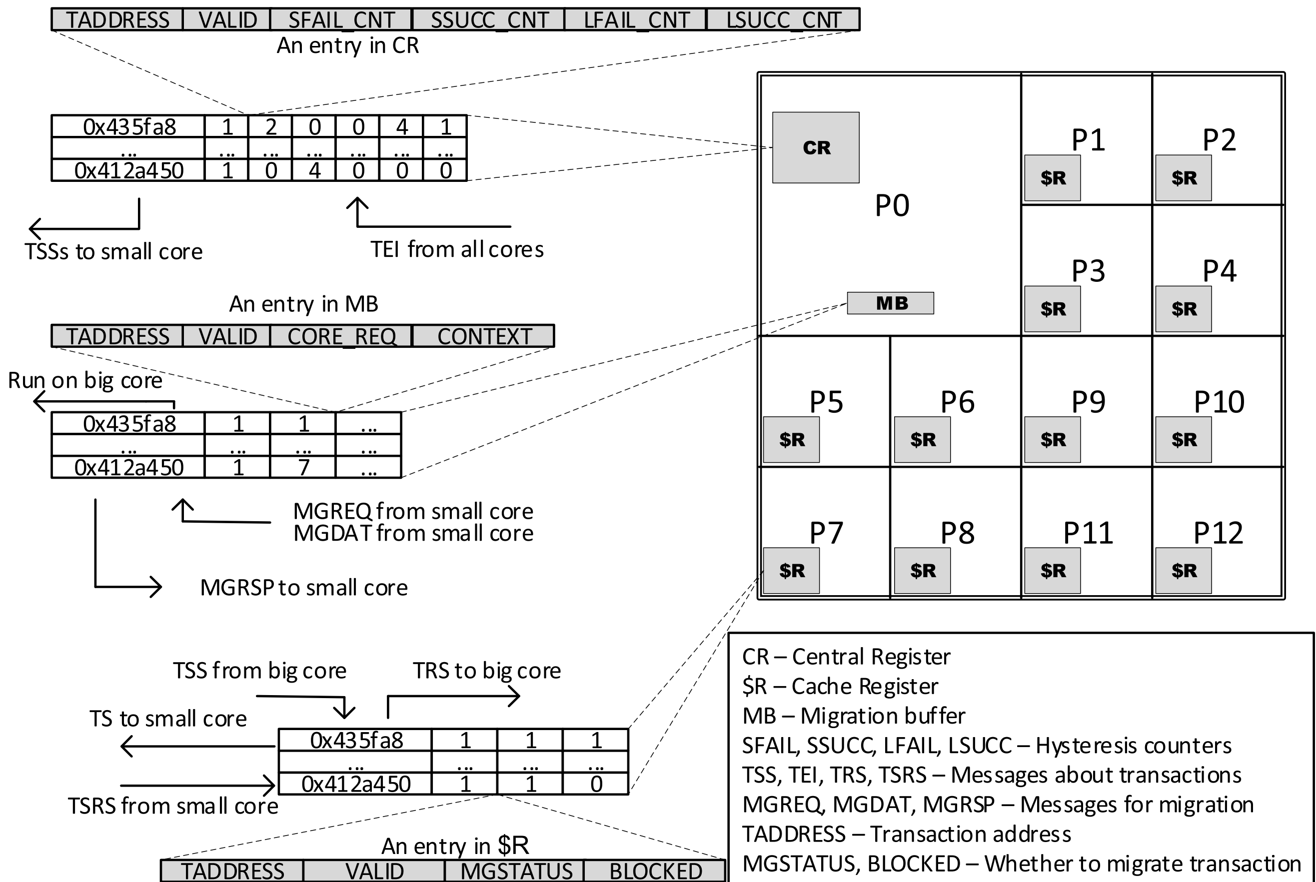

FIGURE 3. Asymmetric multiprocessor with additional units.

that case it send a request to the "big" core for transaction migration. The "small" core starts the transaction execution in case when the transaction history has signaled that the transaction should not be migrated, otherwise the transaction migrates to the "big" core. All communication between the cores occurs through the existing lines for cache data exchange.

Sending a message to the "big" core when starting each transaction and waiting for a response, can significantly compromise execution performance because it is performed before starting any transaction, even those that do not need migration. For that reason, it makes sense to add a small memory to the "small" core, which will store the most up-to-date information on transactions that have recently been executed on a "small" core. We call this small memory the cache register ($R). The $R stores only a unique identifier of the transactions, information showing whether that transaction should be migrated and one bit indicating whether the migration should be blocked locally, as presented in the Figure 3. The unique transaction identifier can be the address of the first instruction of that transaction.

Figure 4a) shows the messages sent and actions that take place in order to migrate transaction from a "small" core to the "big" core. Two cases are presented. One when there is no information about a transaction in the $R. Other is when such information is present, and transaction is successfully migrated.

Just before the start of a transaction, a "small" core requests status (TSRS) from the $R. If the status is not present in the $R, the $R will send request status message (TRS) to the CR. The "small" core will not wait for an answer, but it will start executing the transaction instead. If the status (TS) arrives later, the status will be saved in the $R, so that the next time when the "small" core needs to start this transaction, it will read the information from the $R in order to determine whether the transaction should be migrated or not. Reading status (TSS) from the $R can be performed in a single clock cycle, because the $R is a small memory, so checking the information about the transaction will not cause the significant slowdown. The "small" core requests this information from the $R on execution of the instruction which starts the transaction.

If migration is needed, the instruction that starts the transaction is not executed and the transaction is migrated. No matter if the transaction is migrated or not, the correctness of the program execution is preserved, because the transaction execution on any core has the same effects on data values in the memory, and consequently on the program execution effects. For this reason, executing a transaction on a "small" core until it is determined whether the transaction should

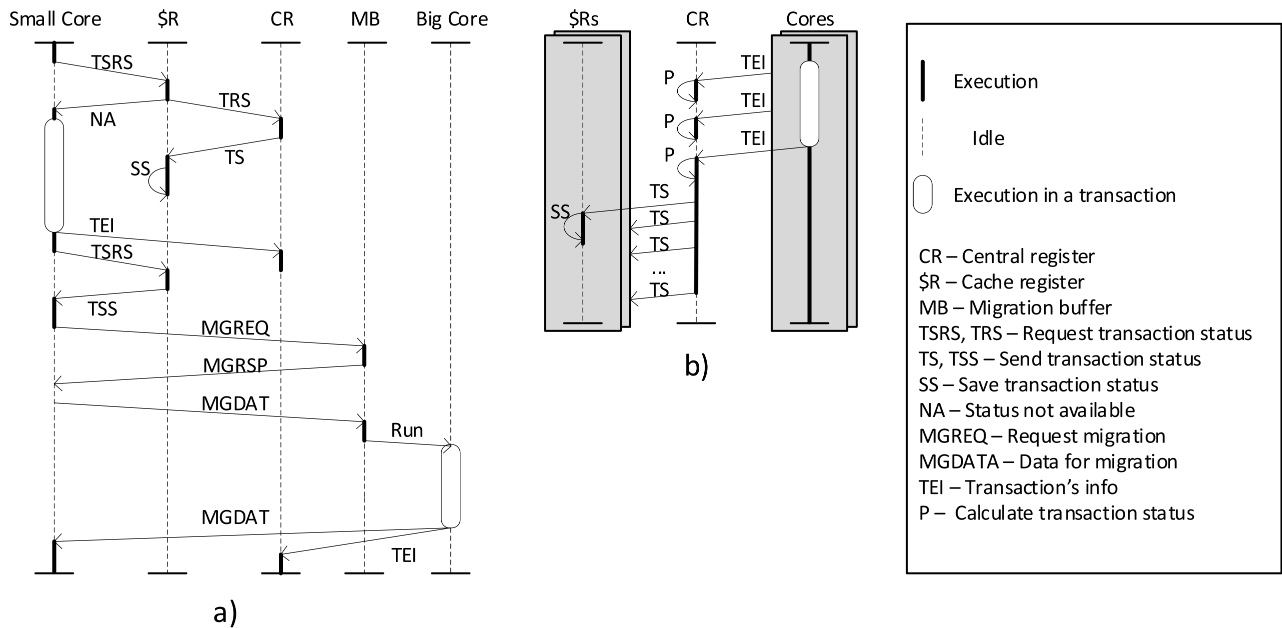

FIGURE 4. **Exchange of messages between the "big" core and a "small" core.**

be migrated can only affect the execution performance. The existing multiprocessors have communication networks between cores that support this way of exchanging messages and sending context. One example of such architecture is Sun Niagara-1 [37], where cores send and receive data from a shared coprocessor for floating point arithmetic.

The "small" core sends a migration request message (MGREQ) when it needs to migrate a transaction. The MGREQ is sent to the migration buffer (MB) in the "big" core. The MB is used for storing a context of transactions that are waiting to be executed on the "big" core. Therefore, if the "big" core is busy, the transaction will wait for migration provided there is enough space in the MB. The MB responds with a migration response message (MGRSP), which may be positive or negative. If it is positive, the "small" core sends the context of execution to the MB via a migration data message (MGDAT). The MB stores the received context. When the "big" core becomes free, it takes the first received context from the MB buffer and starts executing the transaction. If there is no space in the MB buffer, the "small" core receives a negative response and just starts to execute the transaction. Waiting for an answer obviously results in a certain slowdown, but it is inevitable due to limited size of MB buffer. When the "big" core finishes executing the migrated transaction, it sends a MGDAT message to the "small" core, if the execution was successful. If the execution was not successful, the "big" core sends the negative MGRSP and nothing more, because the small core already has the memorized context. When executing the transaction start instruction, the context is saved, so it can be used in the case of transaction cancellation and restart. That context is preserved, although there was migration of the transaction. Figure 4a) shows how cores exchange messages.

Both types of core count instructions during the execution of transactions. When the core reaches the instruction that finishes the transaction (either successful completion or cancellation of the transaction), the core sends an end message (TEI) to the CR, containing information about the success of the transaction, the reason for the completion and its length. The answer to this message is not required, so the core can continue to execute the following instructions. Therefore, the TEI message does not slow down the operation of the core. At the moment of completion, the "small" core sets the bit for local blocking of the transaction. If the transaction was successful, it sets that bit to active value, otherwise to inactive value. If the bit has the active value, the transaction is not migrated, no matter what the global history of the transaction is.

The decision about the migration is made in the CR. After receiving a TEI message, the CR updates the records and determines whether the transaction should be migrated based on it. The Figure 4b) presents the messages exchanged at that moment. Since the information about migration is also stored in $Rs, it is possible that "small" cores do not have up to date information. Outdated information does not affect the correct execution of the program, but it affects the execution performance. In order to provide up-to-date information, when changing that information, the "big" core sends the message TS to all $R, to notify them about the latest change. Those "small" cores that have information about that transaction in their cache register will write the new information. Other cores ignore the message. This way of delivering the latest information may require high bandwidth of communication lines, which is already available in commercial multiprocessors for a long time [38]–[40].

It is necessary to keep records of success, reasons for cancellation and the length of the transaction for the most

TABLE 1. Configuration of simulated multiprocessors.

Parameters	"Big" core	"Small" core
Type	Out-of-Order, 4-way, 2Ghz	In-Order, 2-way, 2GHz
L1 Cache[a]	64KB, 4-way associative, 2-cycles	16KB, 4-way associative, 1-cycle
L2 Cache	256KB, 8-way associative, 6-cycles	64KB, 8-way associative, 2-cycles
L3 Cache	16MB, 32-way associative, 24-cycles	
Network	16B wide, Crossbar topology, 2-cycles each hop	

Configuration[b]	"Big" core	"Small" core	Max. thread number
SMP	0	N	N
AMP	1	N - 4	N - 3
MAMP	1	N - 4	N - 4
MTAMP	1	N - 4	N - 3
FAMP	0	N - 3	N - 4

[a]Only data cache is presented
[b]Chip area for this configuration is N

recent events. Records management is implemented in the same way for all data. Since changes in the state history result in broadcasting messages to all cores, records management should be implemented in such a way to prevent the change in the state of history at each event. One way to do so is to provide hysteresis when determining the state of history, which can be achieved using one counter for each data item. The counters are limited to a minimum and maximum values. When the counter value is maximum, the counter no longer responds to events which increment its value, and when the value is minimal, the counter no longer reacts to events that decrement its value.

The state of the history changes only when the counter reaches a minimum or maximum value. Let us assume that the transaction success counter has a range of 4, the current value is minimal, and the state of history determines that the transaction is not executed successfully. One sequence of events that would lead to a change in the state of history would be: the transaction is executed twice successfully, then once unsuccessfully and then three times successfully.

The range of the counter can be configurable, so that the proper value is set before executing each application. However, in our solution, we assumed the same fixed value for all counters, because the variable range would require complex analysis of applications, which is out of scope of this research.

The proposed implementation does not require excessive chip space. The CR and a $R can have information about a limited number of transactions. If there is no registry space, the replacements are made by an approximation of the algorithm called least recently used (LRU). Some of the entries in the registry can be empty, so it is necessary for each entry to keep one bit showing whether it is valid or not. If the CR has 32 inputs, and the $R for each ‘‘small’’ core has 8 inputs, the required space is less than 3KB, for a multiprocessor with 32 cores. One entry in the migration buffer MB has a thread context consisting of 32 eight-byte registers, transaction address, and core number, which is about 0.25KB. In our implementation, we set the number of inputs to the MB buffer to 4, so the required space is about 1KB. Overall, only around 4KB of memory must be added to the processor.

IV. EXPERIMENTAL ANALYSIS

The evaluation of the proposed solution was performed by simulating the execution of existing representative applications which are used to evaluate transactional memory. In this chapter we describe the implementation of the simulator, its settings, and the test applications that were used. The results are also presented and discussed.

A. EXPERIMENTAL SETTINGS

The simulation was performed using a significantly upgraded Gem5 simulator [41]. That simulator has been selected due to its open code which can be further modified. An important fact about the selected simulator is that its development is supported, and it is used for architectural evaluation by one of the leading processor architecture design companies. Simulation of the processor, memory and communications was performed at the level of processor cycles. Simulation of each test application is performed in isolation, i.e. only that application is executed in the system, from the beginning until the end. All test application threads have their own core for execution, so they cannot be interrupted and there is no thread scheduling. Table 1 shows the characteristics of the ‘‘big’’ and ‘‘small’’ cores, as well as the existing network between them. Both cores are assumed to run on the same frequency, although this is not the case in real implementations.

Usually, the ‘‘big’’ core operates on the higher frequency, which speeds up the execution of the transaction. However, it was found that the simulations with cores operating at different frequencies, have shown that a ‘‘big’’ core works much better than the slower one, compared to the results obtained in real implementations [42]. Therefore, we have taken a more restrictive approach in our simulations, assuming that both

cores operate on the same frequency. The first and the second level caches are private for each core, while the third level cache is shared by all cores.

The "small" core model is based on an Intel Pentium processor [43] which has about 3.3 million transistors. The "big" core is modeled after the Intel Pentium-M core, which has about 14 million transistors [44]. In our analysis, the "big" core does not support multithreaded execution, so its implementation requires fewer transistors. Considering that this assumption is more restrictive for our solution, we will assume that the "big" core is four times larger than the "small" core. In the rest of our analysis we will express the chip size in the number of "small" cores. We will analyze five different multiprocessor configurations: symmetric multiprocessor (SMP) with "small" cores, asymmetric multiprocessor (AMP) with one "big" core and multiple "small" cores, AMP with migration algorithm (MAMP), AMP with migration algorithm and thread execution on the "big" core (MTAMP) and symmetric multiprocessor with migration algorithm (FAMP). The characteristics of each configuration are presented in Table 1. The sizes of all configurations vary from 5 to 36 "small" cores, using the step 4. We start from 5 to provide that each configuration has at least two cores and we end at 36, so that each configuration could execute at least 32 threads.

For simulation analysis, the simulator has been upgraded to support transactional memory, hardware thread migration and all parts of the proposed solution. Transactional memory is implemented so that the records of data accessed in the transaction are kept in the first level cache memory. Conflict detection between transactions is performed using a modified MESI cache coherence protocol [45]. The modification includes new messages that have information on speculative reads and writes during the transaction. The address of data accessed speculatively by another transaction is compared to the set of data addresses speculatively accessed by the current transaction with the aim of determining whether the conflict has occurred. Records of data in the set are kept using a Bloom filter [25] in order to reduce the necessary hardware for its implementation and to improve the efficiency of these sets. However, it should be noticed that false conflicts could arise that would cause unnecessary cancellation of the transaction.

As soon as a conflict is detected, one of the transactions is canceled and restarted. Cancelation is done by invalidating all entries that contained speculatively modified data in the cache memory of the processor which executed the transaction and restoring the processor context to the one that was saved when entering the transaction. The application starts and ends the transaction using transactional memory instructions of the IA-32 architecture [31]. This instruction set is selected to enable compiling test applications more easily, using the existing compilers. Transaction migration is done by using data packets sent from one core to another via communication lines that serve for cache coherence messages transfer. The simulation correctly reflects packet transfer because the buffers and transport lines have size limits, so depending on the amount of messages in the system it can suffer from congestion and the transfer slowdown, which also happens in real systems. The assumed topology is Gem5's default crossbar topology, where each hop has two cycle latency (see Table 1). The context that has to be transferred from one core to another is reduced to the minimal size, necessary to execute a transaction. Having in mind that the set of instructions which can be executed in the transaction is limited, only the basic architectural registers available to the user application have to be transmitted. During migration, the core stops completely, the pipeline is completely flushed, and then the context is saved and sent to another core. Therefore, the migration is a slow operation that, in the case of frequent use, can significantly degrade the system performance. All messages needed for the realization of this solution are implemented using cache coherence packets. Since the directory and cache memory for storing data when transactions need to migrate are associative structures of a small size (the maximum of 32 inputs) they are implemented with a latency of one clock period.

The set of applications selected for this analysis is STAMP [46], because it is a set of applications which has been often used in transactional memory research. Among the existing applications, the ssca2 application was not used in our analysis, because the probability of conflict between transactions is small, so none of the transaction will be selected for migration. Therefore, there is no potential for performance improvement of this application by the proposed solution. The Kmeans application shows similar characteristics, so it is the representative example of the behavior of the proposed solution in the applications with the small probability of conflicts.

The synchronization level in individual tests has not been changed and the tests were used in their original form. However, transaction's locking fallback path is not defined in the original form, so it had to be implemented. The selected solution is that each transaction is repeated for a limited number of times (the limit is the same for all applications). If the maximum number of repetitions is reached, we switch to the synchronization using a global lock [47] in order to guarantee the progress in execution. If the transaction fails due to capacity overflow, the maximum number of repetitions is cut in half. The algorithm used is one of the algorithms described in [48], which provides satisfactory results for almost all applications. Before starting the transaction, each thread will wait for some pseudorandom time, which is generated from a range that grows exponentially with the number of repetitions of that particular transaction. In this way, we try to avoid repetition of scenarios leading to the conflict and transaction cancellation.

Data sets which were used vary in size and they were defined according to the data sets proposed for simulation in [46]. The Kmeans and Vacation applications have two versions, one being with less frequent conflicts, while the other being with more frequent conflicts between transactions.

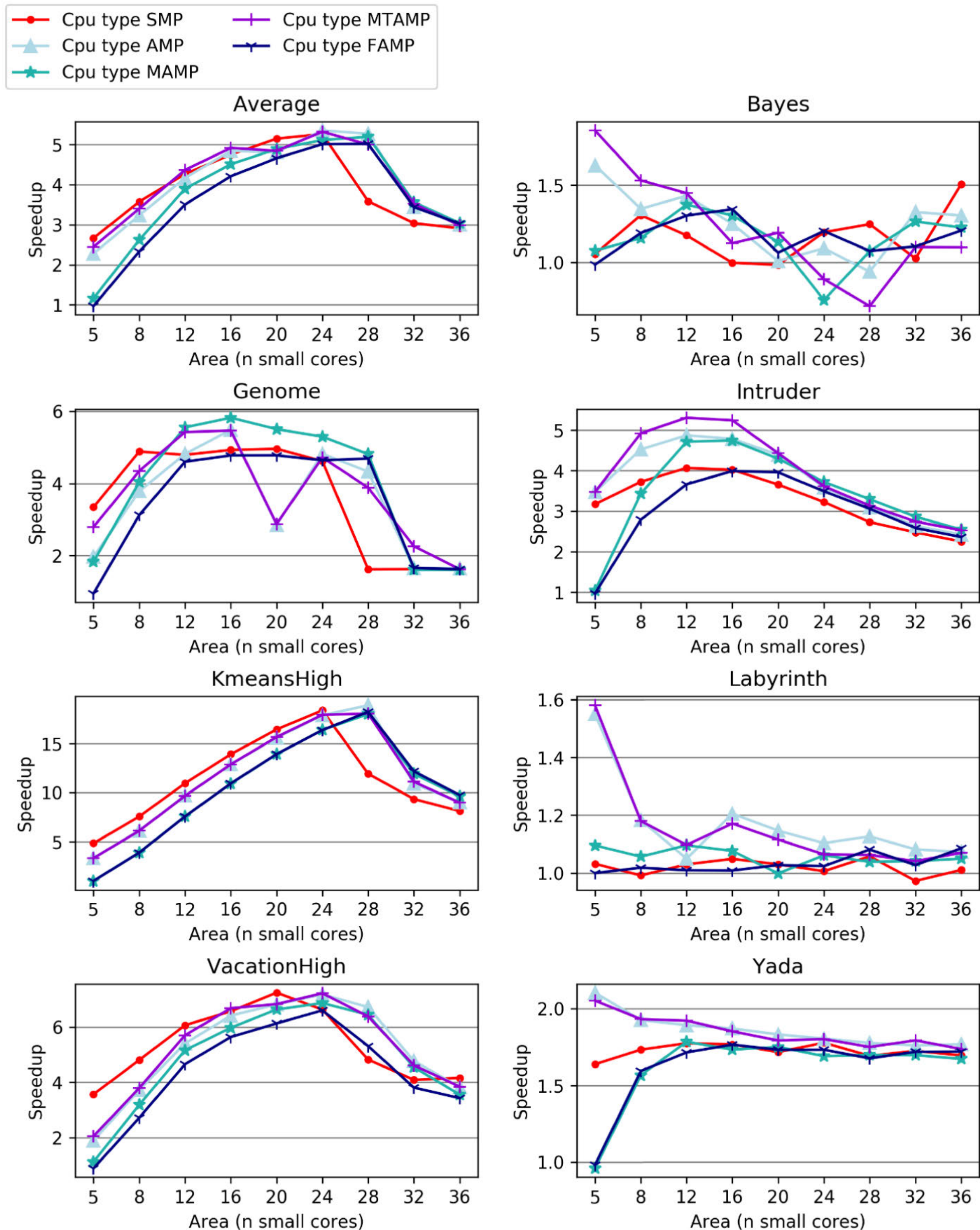

FIGURE 5. Speedup comparison to the system with one "small" core.

In the analysis of the proposed solution we have selected the version with more frequent conflicts because they are more difficult to parallelize. The parameters used for each of the applications are given in Table 2.

B. SIMULATION RESULTS

Figure 5 shows the acceleration achieved by all configurations relative to execution on a single "small" core for all applications. Measurements were performed only for parallel

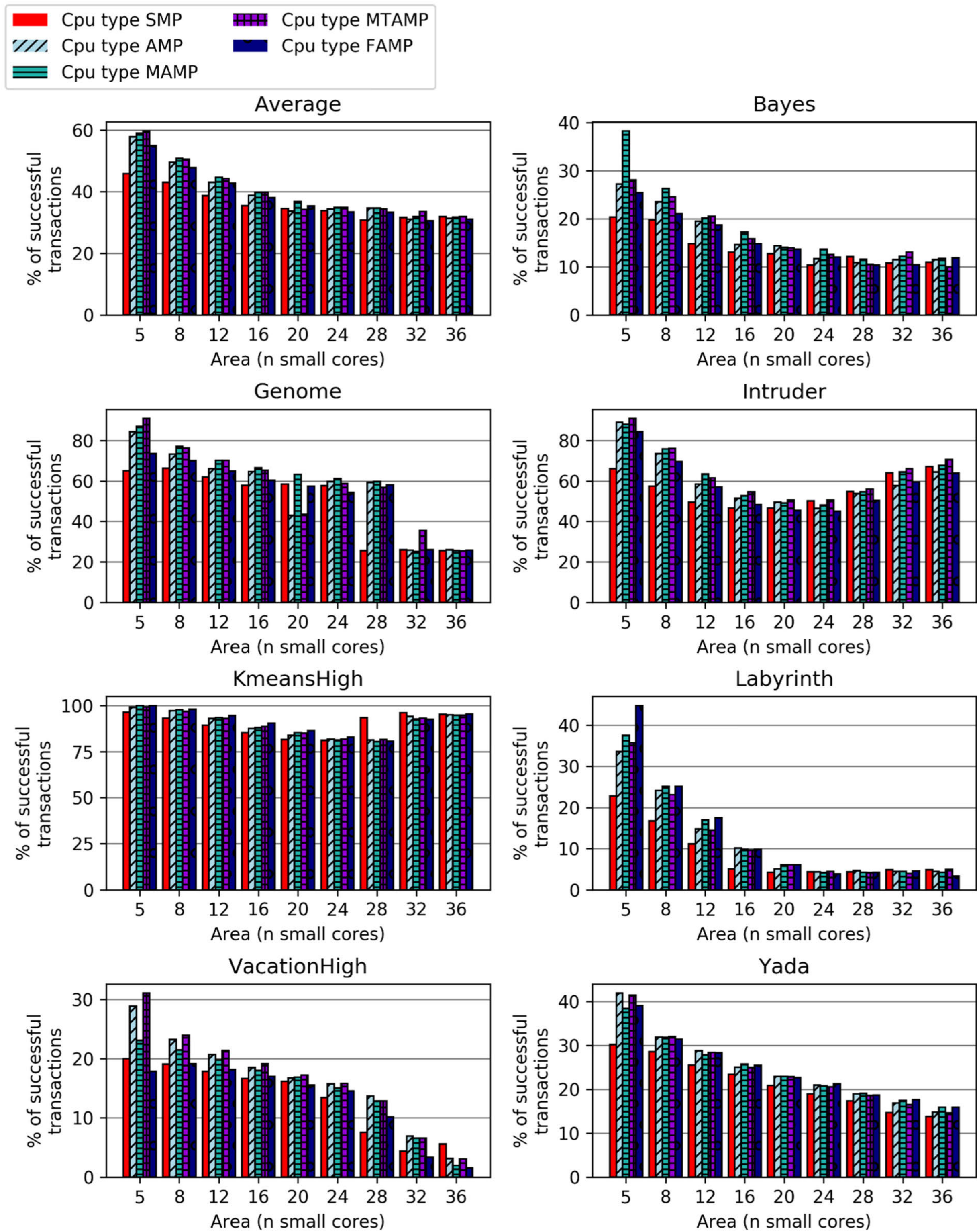

FIGURE 6. **Percentage of successfully executed transactions.**

parts of applications, because the acceleration of the serial part of the application on an asymmetric multiprocessor has already been discussed in [49]–[51]. The same procedures can be applied in this solution. Figure 6 shows the success rate of transactions achieved for all configurations and all applications. The applications were executed with the maximum number of threads allowed for a specific multiprocessor configuration and its size (see Table 1 as a reminder).

TABLE 2. Parameters for starting STAMP applications.

Application	Parameters
Bayes	-v32 -r1024 -n2 -p20 -i2 -e2
Genome	-g256 -s16 -n16384
Intruder	-a10 -l4 -n2048 -s1
Kmeans High	-m15 -n15 -t0.05 -i random-n2048-d16-c16
Labyrinth	-i random-x32-y32-z3-n96
Vacation High	-n4 -q60 -u90 -r16384 -t4096
Yada	-a20 -i 633.2

TABLE 3. MAMP migration success rate.

Application	No. of successful migrations (% of all migrations)	No. of migrations	No. of all transactions
Bayes	412 (18%)	2291 (10.5%)	21748
Genome	3478 (78.7%)	4418 (4.6%)	96079
Intruder	6161 (55.1%)	11179 (6.9%)	162284
Kmeans High	19 (63.3%)	30 (0.0%)	82388
Labyrinth	36 (1.5%)	2430 (27.0%)	8977
Vacation High	4577 (24.6%)	18634 (9.6%)	193499
Yada	245 (1.2%)	20788 (13.0%)	160127

TABLE 4. MTAMP migration success rate.

Application	No. of successful migrations (% of all migrations)	No. of migrations	No. of all transactions
Bayes	181 (8.6%)	2113 (9.1%)	23318
Genome	1901 (50.6%)	3760 (3.9%)	96679
Intruder	11097 (68.4%)	16218 (10.2%)	158742
Kmeans High	17 (74.0%)	23 (0.0%)	82382
Labyrinth	25 (1.0%)	2437 (27.0%)	9035
Vacation High	4299 (28.0%)	15358 (8.3%)	185267
Yada	869 (4.7%)	18556 (11.6%)	160314

Performance slopes for the SMP configurations show that performance grows linearly for the Kmeans and Vacation applications, sub linearly for the Genome and Intruder applications, while it does not scale well with the increase of parallelism for the Bayes, Labyrinth and Yada applications. Those results are consistent with the results previously obtained by other researchers [48], who experimented on real hardware with similarly sized caches and similar organization of transactional memory, so results from our simulator does not deviate from results that are expected on real hardware.

The Kmeans application is highly parallelizable and it has a high rate of transaction success. The speedup grows linearly with the core number increase. Applications of this type do not benefit from our solution since only few transactions are selected for migration by the algorithm. The average number of migrated transactions and their success rates are presented in Tables 3 and 4, for the MAMP and the MTAMP versions of algorithm, respectively. While our solution does not reduce performance substantially (around 2% on average), we recommend turning off migration and allowing one thread to execute on the ‘‘big’’ core since the AMP configuration gives the best result.

The Vacation application is also highly parallelizable as the Kmeans but the number of aborts is high. In this case the number of the aborts does not throttle performance, no matter that the percentage of migration failure is high.

The Yada, Bayes and Labyrinth application have very long transactions with huge read and write sets. This causes low transaction success rate in the existing commercially available HTM solutions due to capacity overflow and high contention. There are no speedup gains with the increase of the number of cores. We consider those applications not suitable for running on HTM since the possible improvement from any solution if hardware is limited by current development of manufacturing technology. For all applications there is a considerable rate of transaction migrations by our algorithms, but the success rate is low, and the ‘‘big’’ core does not reduce transaction execution time enough to lower contention.

The Genome and Intruder applications are not as highly parallelizable as in the case of the Kmeans. However, both applications exhibit higher number of aborts when thread number is increased. These kinds of applications are suitable for our migration algorithms. In Tables 3 and 4 we can see that fairly small percentage of transactions has migrated, but the success rate is high thereby reducing the number of aborts and ultimately increasing the overall performance. The unexpected result is that for the Genome applications better results are accomplished by using the MAMP algorithm, and for the Intruder application by using the MTMAP. The discussion of such a result will be further analyzed in the subsequent section. For the Genome application the configurations AMP and MTAMP with the area of twenty cores has unexpectedly low acceleration. We discard these results due to the unfortunate schedule of the transactions, which is produced by the nature of simulation, i.e., there is no randomness as is the case in a real system.

The FAMP configuration is not intended for implementation on real hardware but for use in simulation, in order to test the proposed algorithm for migration. The idea is to have configurations comprising only ‘‘small’’ cores, while one ‘‘small’’ core in one configuration takes a role of the ‘‘big’’ core in the MAMP configuration, where transactions migrate by means of the algorithm. In this way, different transaction schedules are modeled due to migration, while the transaction execution time remains unmodified. We want to check if the different transaction schedule is the only source of the performance improvement, because if that is the case our solution is overkill, since different transaction schedules can be achieved with much less effort. As the results shows the FAMP configurations have worst performance on average of all types of configurations and that shows that performance improvement in some applications is due to execution of the migrated thread on the ‘‘big’’ core.

C. DISCUSSION

We should emphasize again the previously mentioned fact that the simulated ‘‘big’’ core does not support multithreaded execution. The current state of the technology allows implementations in which the ‘‘big’’ core uses the support for multithreaded execution and the most common implementations

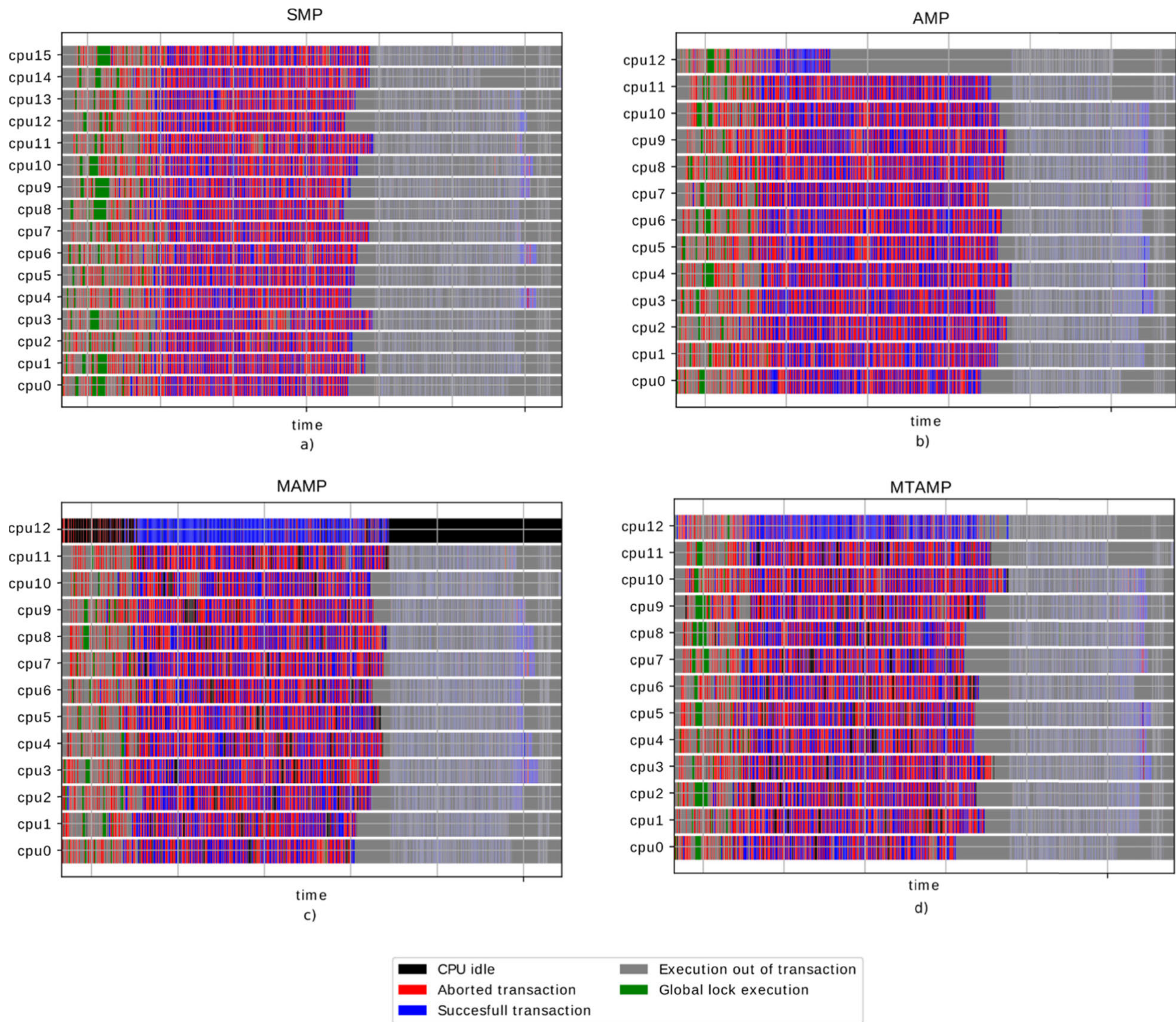

FIGURE 7. STAMP Genome transactions execution on processor with area of 16.

provide support for two threads. In the systems using this kind of a configuration, the ''big'' core would display the increased parallelism, overcoming the problems of some applications caused by the low level of parallelism. That solution would bring the possibility to accelerate more transactions in the application phases that suffer from many conflicts between transactions. It has also been noticed that some transaction migrations turned out to be useless, since they have been interrupted due to conflicts. In order to prevent wasting time on migration, it is possible to make such a ''big'' core the transactions of which have priority over the transactions on ''small'' cores, so that the conflict is resolved by canceling transactions on the ''small'' core. Generally, it is expected that a solution with complex conflict resolution would be significantly more difficult to verify [52], but in this case it should only delay the response to those messages that would eventually cause transactions cancelation. All these elements of the solution should be examined in detail in the future since they could contribute to significant improvements of the proposed solution.

To further investigate why different algorithm made better results on the Genome and the Intruder applications we present the visualization of their execution in the Figures 7 and 8. In these figures, we show for each core what is executed over time (non-transactional execution, successful or unsuccessful transaction execution, failback to locks, and idle). Those two different applications differ in number of distinct phases of executions. The Intruder application has only one phase, while the Genome has three phases. Each phase of the Genome can start when all threads finish with the earlier phase. The ''big'' core in the Genome executes much longer migrations than the Intruder scaled to the overall execution time. The thread on the ''big'' core in the Genome is being delayed due to servicing migration request from other cores. This delay is causing other threads to start executing the next phase later and, consequently, to reduce the performance of

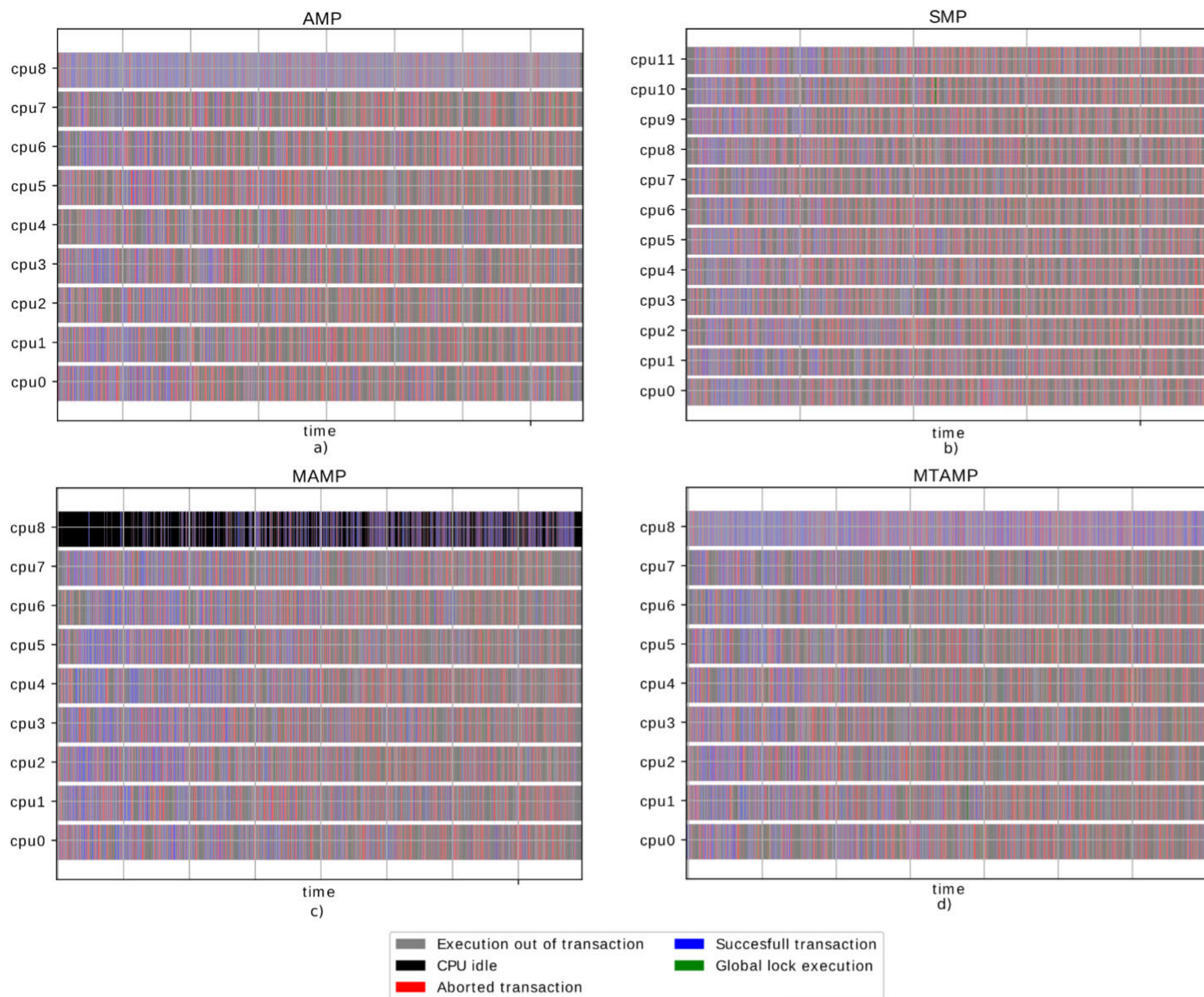

FIGURE 8. STAMP Intruder transactions execution on processor with area of 12.

the whole application. This problem happens only in our fixed setting since only one specific thread is executed on the "big" core and that thread is being constantly delayed. To alleviate this problem, we can schedule another thread to execute on the "big" core.

In our experiments, we have analyzed only single programmed workloads. Our solution can also work in multi-programmed workloads with small modifications. The most important change is to make "small" cores aware of the given process running on the "big" core. The "small" core then needs to skip migration if the process is different from its own. In this way, only one process can benefit from migration, but then migration cost is low since there is no need to change address space on the "big" core. The OS scheduler can then decide which process is more critical and schedule one of its threads to the "big" core. Other necessary changes are related to tracking which transaction belongs to which process (for example by concatenating process id with transaction address in the $R and the CR), to fine-tune the $R and the CR sizes, etc.

We have performed experiments only for the crossbar network topology. Different topologies should be examined in the future with longer communication latencies from core to core.

From the experiments we conclude that our algorithm MTAMP (with the modification of scheduling from time to time different thread on the "big" core) can improve performance for applications that are not highly parallelizable or completely unparallelizable under the available HTM implementation on the Single-ISA multicore processor. For those applications where our algorithm is not suitable, we recommend that the migration algorithm should be turned off. The improvement our algorithm achieves for suitable applications is up to 14% (10% on average) in turnaround time compared to the solutions which do not take advantage of asymmetry for scheduling transactions.

V. RELATED WORK

There is a huge amount of research on Single-ISA asymmetric multicore processors and transactional memory. However,

to the best of our knowledge, there is no research that practically examines running an application on an asymmetric processor with transactional memory support, so we will discuss previous work separately for asymmetric processors and for transactional memory.

A. SINGLE-ISA ASYMMETRIC MULTICORE

A large body of work on Single-ISA asymmetric multicore processors is categorized and described in the extensive survey [1]. The main benefits of using existing solutions are performance gain and energy consumption reduction. Energy consumption analysis is out of the scope of this paper, so we will not discuss solutions the sole purpose of which is to reduce energy consumption. Performance can be improved by migrating work to the right core.

The migration on a fine-grained level is examined in [2], [4]–[6], where it is proposed that bottlenecks that cause other threads to stall should be transferred to a ''big'' core. Those bottlenecks are executions in critical sections or in the lagging threads, which prevent other threads to pass through a barrier. The number of threads which are waiting on critical section executed by another thread is computed by using counters. The lagging thread is determined by the number of threads executing or by estimating the remaining running execution time of the thread by tracking the historical value and current execution time. In [5] authors argue that migration is justifiable only when that part of the thread is on a critical path and they propose metric for determining a measure of criticality. Our work is similar since we migrate on a fine-grain level but it differs from earlier work since the authors have not examined transactions as bottlenecks. The hardware technique we use is similar to those solutions, but it is adapted for transactional memory. Optimization techniques used in those solutions, like ''Data Marshaling'' proposed in [53] used for predictively transferring data from one core to another during migration to reduce the number of cache's misses, can be used to complement our solution or any other cache optimization [54].

The migration on a coarse-grained level is examined in previous research, where it is proposed that the whole threads should be migrated to the right core, mostly using OS scheduler. With the inclusion of fairness consideration the performance of multiprogram workload can also be improved by utilizing software. In [9], [10] scheduling is based on speedup that is expected on a ''big'' core. If some applications are lagging, their threads are swapped to a ''big'' core. The scheduling in [7], [8] is done by modeling performance estimates using machine learning, i.e., linear regression on the values of performance counters. The machine learning is frequently used to achieve better performance in systems with transactional memory [55]. The focus in [11]–[13] is on the fairness of scheduling. Formulas for calculating fairness based on time or execution progress on each type of core are presented. When some thread has the lower value than other threads it is scheduled to the other type of core. In [11], [12] authors suggest that scheduling can be done in hardware, but the work lacks detailed description. We consider these researches orthogonal to our own because these software schedulers can be used together with our solution, especially to solve problem of running multiprogram workload, which we have not considered in this paper.

In addition to conventional applications, migration is examined for task-parallel applications in the context of asymmetric multicore processors. Task parallel application is an application which consists of multiple tasks executed on multiple worker threads. A task can depend on other tasks and only when the dependent tasks are executed the task can proceed with execution. Some existing solutions [56], [57] schedule tasks based on criticality on big cores (by analyzing task dependency graph) and based on computational needs of the tasks. Other solutions [58], [59] propose work stealing when some condition is met, like when the number of executing tasks is low. Work stealing represents migrating tasks from the small to the big cores. Our work differs in that it proposes a performance improvement for transactional memory-based applications.

B. TRANSACTIONAL MEMORY

The existing research on transactional memory is vast and many of the existing solutions can be used as a base for our solution, so we solely consider research on transaction scheduling since it is most similar to our work. The scheduling of transactions includes collecting data about transaction conflicts and generating decision based on that data whether to delay transaction or to execute it. Some solutions require hardware changes to collect necessary data, while recent solutions use only limited data provided by commercially available processors with HTM support.

The scheduling algorithms proposed in [19]–[21] require programming interface modifications to get precise information about transactions (either conflicting transactions or transactions' access trace). Since changing programming interface will be rare in future versions of commercially available processors with HTM support we consider our solution to be more feasible.

Simple scheduling is proposed in [14], where contention is monitored by keeping track of abort rate. In cases when the abort rate exceeds a preset threshold, transactions are being delayed. Reducing the number of transactions currently executing in [18] is done by using auxiliary locks in a transaction. This solution assumes that transaction code is built from an already existing lock-based code and programmer-added locks are used as auxiliary locks. In [15] when transaction commits or aborts, data are collected which other transactions are active. If it is a commit that means that active transactions do not conflict and can be executed in parallel, otherwise they should not be scheduled at the same time. Based on the information each transaction accesses a lock which is used as scheduling manager. Similarly, in [16], [17] the same principle is used but instead of locks, the number of queues is made variable, so it is increased or decreased based on commit/abort rate. Our research can be considered similar,

since we also reduce parallelism (recall several migrated transactions can wait in queue to execute on the "big" core). However, the difference is that we try to preserve parallelism as much as possible and avoid conflicts by reducing time window in which conflict can happen. On the other hand, our solution can be used together with these solutions without any modification.

VI. CONCLUSION

In this paper we have proposed and investigated a system with the Single-ISA asymmetric multiprocessor which supports hardware transactional memory. We have presented methods and techniques on which we have built the prototype for the M-HTM system. The system performs transaction migration using an algorithm fully implemented in hardware. Two variants of the algorithm were proposed: MAMP and MTAMP. The MTAMP achieved better results. Using simulation, we have investigated the impact of the migrations on the performance of the several applications from the STAMP benchmark test suit. Our solution achieved higher performance for the applications that are suitable for migration, while for the other applications it achieved comparable performance. The suitable applications are the ones which are parallelizable but not to a high extent and which do not have short transactions. We proposed and described the solution for an architecture, which is underpowered, so we consider our results conservative. In the system which has more "big" cores with simultaneous multithreading we expect that our solution will achieve better results. We have discussed how our algorithm could be modified for the purpose of such a system, but further details will be presented, and improvements will be investigated in our future work.

REFERENCES

[1] S. Mittal, "A survey of techniques for architecting and managing asymmetric multicore processors," *ACM Comput. Surv.*, vol. 48, no. 3, pp. 1–38, Feb. 2016, doi: 10.1145/2856125.

[2] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," in *Proc. 14th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2009, p. 253, doi: 10.1145/1508244.1508274.

[3] M. Herlihy and J. E. B. Moss, "Transactional memory," *ACM SIGARCH Comput. Archit. News*, vol. 21, no. 2, pp. 289–300, May 1993, doi: 10.1145/173682.165164.

[4] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck identification and scheduling in multithreaded applications," in *Proc. 17th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, Apr. 2012, vol. 40, no. 1, p. 223, doi: 10.1145/2150976.2151001.

[5] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Utility-based acceleration of multithreaded applications on asymmetric CMPs," in *Proc. 40th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2013, vol. 41, no. 3, pp. 154–165, doi: 10.1145/2485922.2485936.

[6] N. B. Lakshminarayana, J. Lee, and H. Kim, "Age based scheduling for asymmetric multiprocessors," in *Proc. Conf. High Perform. Comput. Netw., Storage Anal. (SC)*, 2009, p. 1, doi: 10.1145/1654059.1654085.

[7] T. Yu, R. Zhong, V. Janjic, P. Petoumenos, J. Zhai, H. Leather, and J. Thomson, "Collaborative heterogeneity-aware OS scheduler for asymmetric multicore processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 5, pp. 1224–1237, May 2021, doi: 10.1109/TPDS.2020.3045279.

[8] I. Jibaja, T. Cao, S. M. Blackburn, and K. S. McKinley, "Portable performance on asymmetric multicore processors," in *Proc. Int. Symp. Code Gener. Optim.*, no. 1, Feb. 2016, pp. 24–35, doi: 10.1145/2854038.2854047.

[9] J. C. Saez, A. Pousa, F. Castro, D. Chaver, and M. Prieto-Matias, "Towards completely fair scheduling on asymmetric single-ISA multicore processors," *J. Parallel Distrib. Comput.*, vol. 102, pp. 115–131, Apr. 2017, doi: 10.1016/j.jpdc.2016.12.011.

[10] A. Garcia-Garcia, J. C. Saez, and M. Prieto-Matias, "Contention-aware fair scheduling for asymmetric single-ISA multicore systems," *IEEE Trans. Comput.*, vol. 67, no. 12, pp. 1703–1719, Dec. 2018, doi: 10.1109/TC.2018.2836418.

[11] C. Kim and J. Huh, "Exploring the design space of fair scheduling supports for asymmetric multicore systems," *IEEE Trans. Comput.*, vol. 67, no. 8, pp. 1136–1152, Aug. 2018, doi: 10.1109/TC.2018.2796077.

[12] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *Proc. 39th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2012, pp. 213–224.

[13] N. Markovic, D. Nemirovsky, O. Unsal, M. Valero, and A. Crista, "Thread lock section-aware scheduling on asymmetric single-ISA multi-core," *IEEE Comput. Archit. Lett.*, vol. 14, no. 2, pp. 160–163, Jul. 2015, doi: 10.1109/LCA.2014.2357805.

[14] R. M. Yoo and H.-H.-S. Lee, "Adaptive transaction scheduling for transactional memory systems," in *Proc. 20th Annu. Symp. Parallelism Algorithms Archit. (SPAA)*, May 2008, p. 169, doi: 10.1145/1378533.1378564.

[15] N. Diegues, P. Romano, and S. Garbatov, "Seer: Probabilistic scheduling for hardware transactional memory," in *Proc. 27th ACM Symp. Parallelism Algorithms Archit.*, Jun. 2015, pp. 224–233, doi: 10.1145/2755573.2755578.

[16] M. Mohamedin, R. Palmieri, and B. Ravindran, "On scheduling best-effort HTM transactions," in *Proc. 27th ACM Symp. Parallelism Algorithms Archit.*, Jun. 2015, pp. 74–76, doi: 10.1145/2755573.2755612.

[17] Z. Chen, A. Hassan, M. J. Kishi, J. Nelson, and R. Palmieri, "HATS: Hardware-assisted transaction scheduler," in *Proc. Leibniz Int. Informat. (LIPIcs)*, 2020, vol. 153, no. 10, pp. 1–10, doi: 10.4230/LIPIcs.OPODIS.2019.10.

[18] Y. Afek, A. Levy, and A. Morrison, "Software-improved hardware lock elision," in *Proc. ACM Symp. Princ. Distrib. Comput. (PODC)*, 2014, pp. 212–221, doi: 10.1145/2611462.2611482.

[19] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, A. Bhandari, and E. Witchel, "TxLinux: Using and managing hardware transactional memory in an operating system," in *Proc. 21st ACM SIGOPS Symp. Oper. Syst. Princ.*, 2007, vol. 41, no. 6, pp. 87–101, doi: 10.1145/1294261.1294271.

[20] M. Ansari, B. Khan, M. Luján, C. Kotselidis, C. Kirkham, and I. Watson, "Improving performance by reducing aborts in hardware transactional memory," in *Proc. Int. Conf. High-Perform. Embedded Archit. Compil.*, in Lecture Notes in Computer Science: Including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics, vol. 5952, 2010, pp. 35–49.

[21] G. Blake, R. G. Dreslinski, and T. Mudge, "Bloom filter guided transaction scheduling," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, Feb. 2011, pp. 75–86, doi: 10.1109/HPCA.2011.5749718.

[22] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, "Bulk disambiguation of speculative threads in multiprocessors," in *Proc. 33rd Int. Symp. Comput. Archit. (ISCA)*, 2006, pp. 227–238, doi: 10.1109/ISCA.2006.13.

[23] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing transactional memory," in *Proc. 32nd Int. Symp. Comput. Archit. (ISCA)*, 2005, pp. 494–505, doi: 10.1109/ISCA.2005.54.

[24] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling hardware transactional memory from caches," in *Proc. IEEE 13th Int. Symp. High Perform. Comput. Archit.*, Feb. 2007, pp. 261–272, doi: 10.1109/HPCA.2007.346204.

[25] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970, doi: 10.1145/362686.362692.

[26] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *Proc. 11th Int. Symp. High-Perform. Comput. Archit.*, 2005, no. 1, pp. 316–327, doi: 10.1109/HPCA.2005.41.

[27] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. Van Biesbrouck, G. Pokam, B. Calder, and O. Colavin, "Unbounded page-based transactional memory," in *Proc. 12th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS-XII)*, 2006, p. 347, doi: 10.1145/1168857.1168901.

[28] J. Chung, C. C. Minh, A. McDonald, T. Skare, H. Chafi, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun, ''Tradeoffs in transactional memory virtualization,'' in *Proc. 12th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS-XII)*, Oct. 2006, vol. 41, no. 11, p. 371, doi: 10.1145/1168857.1168903.

[29] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, ''LogTM: Log-based transactional memory,'' in *Proc. 12th Int. Symp. High-Perform. Comput. Archit.*, 2006, pp. 258–269, doi: 10.1109/HPCA.2006.1598134.

[30] C. Blundell, E. C. Lewis, and M. M. Martin. (2006). *Unrestricted Transactional Memory: Supporting I/O and System Calls Within Transactions*. [Online]. Available: http://repository.upenn.edu/cis_reports/130/

[31] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corp., Santa Clara, CA, USA, 2018, pp. 385–392.

[32] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, ''Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction,'' in *Proc. MICRO*, Dec. 2003, pp. 81–92. Accessed: Jul. 16, 2014. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1253185

[33] P. Greenhalgh. (2011). Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7. ARM. Accessed: Oct. 15, 2019. [Online]. Available: https://www.eetimes.com/big-little-processing-with-arm-cortex-a15-cortex-a7/#

[34] H. Chung, M. Kang, and H.-D. Cho. (2013). Heterogeneous multiprocessing solution of Exynos 5 Octa with ARM big.LITTLETM technology. Samsung. Accessed: Nov. 03, 2019. [Online]. Available: https://s3.ap-northeast-2.amazonaws.com/global.semi.static/Heterogeneous_Multi-Processing_Solution_of_Exynos_5_Octa_with_ARM_bigLITTLE_Technology.pdf

[35] M. Becchi and P. Crowley, ''Dynamic thread assignment on heterogeneous multiprocessor architectures,'' in *Proc. 3rd Conf. Comput. Frontiers (CF)*, vol. 10, 2006, p. 29, doi: 10.1145/1128022.1128029.

[36] K. C. Yeager, ''The Mips R10000 superscalar microprocessor,'' *IEEE Micro*, vol. 16, no. 2, pp. 28–41, Apr. 1996, doi: 10.1109/40.491460.

[37] P. Kongetira, K. Aingaran, and K. Olukotun, ''Niagara: A 32-way multithreaded sparc processor,'' *IEEE Micro*, vol. 25, no. 2, pp. 21–29, Mar. 2005, doi: 10.1109/MM.2005.35.

[38] A. Ahmed, P. Conway, B. Hughes, and F. Weber, ''AMD opteron shared memory MP systems,'' in *Proc. 14th HotChips Symp.*, 2002, pp. 1–30.

[39] D. C. Bossen, J. M. Tendler, and K. Reick, ''Power4 system design for high reliability,'' *IEEE Micro*, vol. 22, no. 2, pp. 16–24, Mar. 2002, doi: 10.1109/MM.2002.997876.

[40] R. Kalla, B. Sinharoy, and J. Tendler, ''Simultaneous multi-threading implementation in POWER5,'' in *Proc. Conf. Rec. Hot Chips Symp.*, 2003.

[41] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, ''The gem5 simulator,'' *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, May 2011, doi: 10.1145/2024716.2024718.

[42] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, ''The impact of performance asymmetry in emerging multicore architectures,'' in *Proc. 32nd Int. Symp. Comput. Archit. (ISCA)*, May 2005, vol. 33, no. 2, pp. 506–517, doi: 10.1109/ISCA.2005.51.

[43] ''Pentium processor data book,'' in *Pentium Processor User's Manual*, vol. 1. Santa Clara, CA, USA: Intel Corporation, 1993.

[44] S. Gochman, ''The Intel Pentium M processor: Microarchitecture and performance,'' *Intel Technol. J.*, vol. 7, no. 2, pp. 1–18, 2003.

[45] M. S. Papamarcos and J. H. Patel, ''A low-overhead coherence solution for multiprocessors with private cache memories,'' in *Proc. 11th Annu. Int. Symp. Comput. Archit. (ISCA)*, 1984, pp. 348–354, doi: 10.1145/800015.808204.

[46] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, ''STAMP: Stanford transactional applications for multi-processing,'' in *Proc. IEEE Int. Symp. Workload Characterization*, Oct. 2008, pp. 35–46, doi: 10.1109/IISWC.2008.4636089.

[47] *Programming With Intel Transactional Synchronization Extensions: Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corp., Santa Clara, CA, USA, 2016, pp. 385–392.

[48] N. Diegues and P. Romano, ''Self-tuning Intel restricted transactional memory,'' *Parallel Comput.*, vol. 50, pp. 25–52, Dec. 2015, doi: 10.1016/j.parco.2015.10.001.

[49] M. D. Hill and M. R. Marty, ''Amdahl's law in the multicore era,'' *Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008, doi: 10.1109/MC.2008.209.

[50] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, ''Heterogeneous chip multiprocessors,'' *Computer*, vol. 38, no. 11, pp. 32–38, Nov. 2005, doi: 10.1109/MC.2005.379.

[51] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, and E. Ayguade, ''Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors,'' *IEEE Comput. Archit. Lett.*, vol. 5, no. 1, p. 4, Jan. 2006, doi: 10.1109/L-CA.2006.6.

[52] R. Quislant, E. Gutierrez, E. L. Zapata, and O. Plata, ''Conflict detection in hardware transactional memory,'' in *Transactional Memory. Foundations, Algorithms, Tools, and Applications*. Cham, Switzerland: Springer, 2015, pp. 127–149.

[53] M. A. Suleman, O. Mutlu, J. A. Joao, Khubaib, and Y. N. Patt, ''Data marshaling for multi-core architectures,'' in *Proc. 37th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2010, p. 441, doi: 10.1145/1815961.1816020.

[54] Z. Sustran, G. Rakocevic, and V. Milutinovic, ''Dual data cache systems,'' in *Advances in Computers*, vol. 96. Amsterdam, The Netherlands: Elsevier, 2015, pp. 187–233, doi: 10.1016/bs.adcom.2014.11.001.

[55] I. Vurdelja, Ž. Šuštran, J. Protić, and D. Draskovic, ''Survey of machine learning application in transactional memory,'' in *Proc. 28th Telecommun. Forum (TELFOR)*, Nov. 2020, pp. 1–4, doi: 10.1109/TELFOR51502.2020.9306547.

[56] M. Han, J. Park, and W. Baek, ''Design and implementation of a Criticality- and heterogeneity-aware runtime system for task-parallel applications,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 5, pp. 1117–1132, May 2021, doi: 10.1109/TPDS.2020.3031911.

[57] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero, ''Criticality-aware dynamic task scheduling for heterogeneous architectures,'' in *Proc. 29th ACM Int. Conf. Supercomput.*, Jun. 2015, pp. 329–338, doi: 10.1145/2751205.2751235.

[58] Q. Chen, Y. Chen, Z. Huang, and M. Guo, ''WATS: Workload-aware task scheduling in asymmetric multi-core architectures,'' in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2012, pp. 249–260, doi: 10.1109/IPDPS.2012.32.

[59] C. Torng, M. Wang, and C. Batten, ''Asymmetry-aware work-stealing runtimes,'' in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 40–52, doi: 10.1109/ISCA.2016.14.

ZIVOJIN SUSTRAN received the B.Sc. and M.Sc. degrees in electrical engineering and computing from the School of Electrical Engineering, University of Belgrade, Serbia, in 2010 and 2012, respectively, where he is currently pursuing the Ph.D. degree. He is currently a Teaching Assistant with the School of Electrical Engineering, University of Belgrade. He has been involved in the research and development of hardware and software solutions in industry and academia for ten years, with expertise in computer architecture, cache memory design, systems programming, operating systems, and FPGA acceleration. He has coauthored two journal articles and gave talks at conferences in Europe. His current research interests include cache coherence and shared memory algorithms, hardware transactional memory, multicore architectures, and with special emphasis on asymmetric multiprocessors.

JELICA PROTIC received the Ph.D. degree in electrical engineering from the University of Belgrade. She is currently a Full Professor of computer engineering and informatics with the School of Electrical Engineering, University of Belgrade. She was a Principal Designer in pioneer projects of networking proprietary industrial computers. With Milo Tomasevic and Veljko Milutinovic, she coauthored *Distributed Shared Memory: Concepts and Systems* (IEEE CS Press, 1997) and presented numerous pre-conference tutorials on this subject. She also conducted research in the domain of wireless sensor networks. She has long term experience in teaching a diversity of courses in programming languages and the development of various educational software tools. Her research interests include distributed systems, consistency models, computer networks, and all aspects of computer-based quantitative performance analysis and modeling.