

SoC-TM: Integrated HW/SW Support for Transactional Memory Programming on Embedded MPSoCs

Cesare Ferri¹
cesare_ferri@brown.edu

Tali Moreshet³
tali@swarthmore.edu

Andrea Marongiu²
a.marongiu@unibo.it

R. Iris Bahar¹
iris_bahar@brown.edu

Maurice Herlihy⁴
herlihy@cs.brown.edu

Benjamin Lipton³
blipto1@swarthmore.edu

Luca Benini²
luca.benini@unibo.it

¹School of Engineering, Brown University, Providence, RI 02912, United States

²DEIS, University of Bologna, Viale Risorgimento 2, 40138 Bologna, Italy

³Engineering Department, Swarthmore College, Swarthmore, PA 19081, United States

⁴Computer Science Department, Brown University, Providence, RI 02912, United States

ABSTRACT

Two overriding concerns in the development of embedded MPSoCs are ease of programming and hardware complexity. In this paper we present *SoC-TM*, an integrated HW/SW solution for transactional programming on embedded MP-SoCs. Our proposal leverages a Hardware Transactional Memory (HTM) design, based on a dedicated HW module for conflict management, whose functionality is exposed to the software through compiler directives, implemented as an extension to the popular OpenMP programming model. To further improve ease of programming, our framework supports speculative parallelism, thanks to the ability of enforcing a given commit order in hardware. Our experimental results confirm that *SoC-TM* is a viable and cost-effective solution for embedded MPSoCs, in terms of energy, performance and productivity.

Categories and Subject Descriptors

B.3.2 [Memory Structures]: Design Styles—*Cache Memory, Shared Memory*; D.1.3 [Concurrent Programming]: [Parallel Programming]; D.3.3 [Language Constructs and Features]: [Concurrent Programming Structures]

General Terms

Design, Experimentation, Performance

Keywords

Transactional Memory, OpenMP, Speculative parallelism, MPSoC

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CODES+ISSS'11, October 9–14, 2011, Taipei, Taiwan.

Copyright 2011 ACM 978-1-4503-0715-4/11/10 ...\$10.00.

1. INTRODUCTION

In an effort to meet an ever increasing demand for performance within tight energy constraints, embedded systems have embraced the multi-core design paradigm, where an increasing number of simple, low-power cores is integrated on the same chip, communicating through multi-level on-chip memory hierarchies. Keeping hardware simple is key to reducing design complexity and energy consumption; thus the burden of fully harnessing the computational power that these systems can deliver is shifted more to the software side, which becomes the critical path in embedded system development. The memory model of a MPSoC defines the responsibilities and challenges of the programmer with respect to management of data. For distributed memory MP-SoCs, where each core features private tightly coupled storage, programmers are responsible for orchestrating efficient data movement before computation can start. Programming shared memory MPSoCs poses a different challenge, since there is the necessity of synchronizing cores so as to prevent memory inconsistencies when data races arise.

The shared memory paradigm is widely adopted in embedded MPSoC designs, since it provides an easy-to-understand abstraction of memory resources: a single address space, to which programmers are accustomed. It is therefore of the utmost importance to simplify thread synchronization in shared memory programs.

To achieve high performance, fine-grained locking techniques may be used to minimize sequentialization of core activities, but this requires deep understanding of the target application and complicated debug processes. Coarse-grained locking is easier to use, but can not extract high degrees of parallelism.

Transactional Memory (TM) has emerged as a promising approach to address the difficulties of parallel programming for shared memory multi-cores in the domain of general-purpose computing. Transactions are as easy to use as coarse-grained locks — programmers only have to identify the boundaries of critical code regions — but promise the same performance of the finest locking granularity. Concurrent transactional threads optimistically execute in parallel,

as if no data conflict may possibly arise. If a data conflict takes place, the TM system detects it, and appropriately restarts one of the conflicting transactions ensuring that its changes to memory are not visible to the rest of the system. Several TM systems have been proposed, based on hardware, software, or hybrid techniques [7]. While transactional memory has been extensively studied for the general-purpose computing domain, there have been relatively few works that consider adopting TM in the embedded domain (e.g., [5, 9, 13]). However, any practical TM design for embedded systems must emphasize simplicity; complex hardware designs that require extensive changes to established protocols (i.e., cache coherency), will likely be too costly to adapt. Similarly, TM must be integrated into practical and familiar programming environments, with simple programming abstractions, in order to easily use it in an embedded domain. In the general-purpose domain a first attempt in this direction was made with the OpenTM programming framework, which extends the popular OpenMP API for shared-memory systems with compiler directives to express TM programming patterns [1].

In this paper we present *SoC-TM*, an integrated HW/SW solution for transactional programming on embedded MP-SoCs. At the heart of our proposal sits a Hardware Transactional Memory (HTM) design that requires only a few minor modifications to the cache system, plus a dedicated HW component – the *Bloom module* – which is composed of a collection of Bloom filters and is responsible for keeping track of the history of each transaction and managing conflicts. While other works have also proposed the use of Bloom filters as a means of managing conflicts among transactions (e.g., [22]), what we propose is a centralized unit that is fully implemented in hardware. This approach has the advantage of being very lightweight and fast, requiring minimal changes to the cache protocol, and removing the need for each core to make individual decisions on transaction conflict management.

Ease of use and transparency of the internal functionality are first-class design requirements of the *SoC-TM* system. As such, application developers are not meant to cope directly with low-level transactional programming. Instead, in our system transactional features are triggered through the use of a custom set of compiler directives, implemented as an extension to the popular OpenMP programming model. Our enhanced compiler transforms the annotated program so as to interact with our runtime system, where low-level APIs are transparently used.

To further improve ease of programming, our framework supports speculative task- and data-level parallelism. Specifically, loops which may carry cross-iteration dependencies (non-DOALL) can be annotated as parallel loops. The underlying TM system ensures that, in case a real dependence arises, the original sequential program semantics is preserved. In our *SoC-TM* system this can be achieved by forcing transactions to commit in program order, thanks to specific hardware support for prioritized commit that we provide. At the program level, this feature can be leveraged by the OpenMP worksharing constructs to generate parallel transactions holding speculative workloads (tasks or loop iterations). The compiler properly programs the Bloom module to commit speculative transactions in order.

Summarizing, we make the following contributions:

- We present the Bloom module, a centralized hardware

module providing lightweight support to conflict resolution and ordered commits, along with a low-level programming API for conveniently programming it.

- We transparently integrate the Bloom module within the OpenMP programming model, extended with language features for transactional programming.
- We provide additional extensions to support speculative parallelization of non DOALL loops.

Our experimental results confirm that *SoC-TM* is a viable and cost-effective solution for embedded MPSoCs, in terms of energy, performance and productivity.

The rest of the paper is organized as follows: An overview of the target platform and the *SoC-TM* system is described in Section 2. A detailed description of the hardware and software support is provided in Sections 3 and 4. Experimental setup and results are described in Section 5, while Section 6 concludes the paper and shows future research directions.

2. SYSTEM OVERVIEW

2.1 Architectural Template

To scale to many-core systems, embedded MPSoCs are increasingly relying on a design paradigm where a general purpose *host* processor controls an on-chip GPU-like accelerator, made of tens to hundreds of very simple homogeneous cores [19] [16]. Our focus in this paper is on the host processor subsystem. Current designs feature a dual-core system. Quad-cores are underway, and next-generation products will be based on eight-core designs.

Fig. 1 depicts the block diagram of the target platform considered in this work. It features a configurable num-

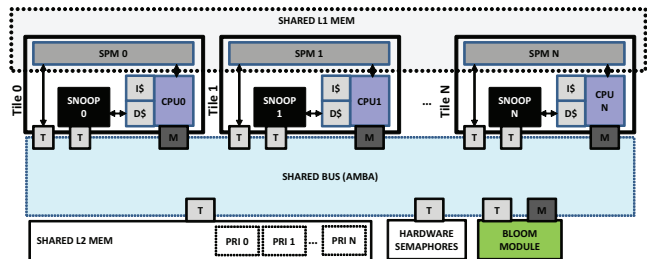


Figure 1: Target architecture

ber (up to 8) of RISC-like cores, interconnected through a shared bus (AMBA). Each core has private L1 instruction and data caches, the latter being kept globally coherent through per-core snoop devices which implement a MESI coherence protocol. The shared memory is organized as a two-level hierarchy, adhering to the Partitioned Global Address Space (PGAS) paradigm. More specifically, our MPSoC features several distinct physical memory banks, globally visible throughout the system. Each core has a small L1 local scratchpad (SPM), which can be accessed without traversing the system interconnect. Remote SPMs can also be accessed directly, but corresponding transactions travel through the bus, and thus are subject to higher latencies. The overall L1 shared memory can be seen as the sum of all SPMs, and is globally non-coherent. This means that its addresses are not cacheable, and it is explicitly managed by software. We use L1 shared memory to optimize the implementation of

runtime services (e.g., thread scheduling, barrier synchronization), but expert programmers can also explicitly use it to optimize critical parts of their applications.

L2 shared memory physically consists of a single device, but we logically partition it into a large, shared segment, plus small “private” segments for each core. Addresses belonging to the logically shared chunk are cacheable and globally coherent. “Private” segments are also cacheable, but their addresses are not involved in coherency traffic.

Explicitly separating shared data from private data is done automatically by our compiler (Sec. 4), and has the advantage of making conflict management in the Bloom module much lighter (Sec. 3).

Finally, synchronization among cores is achieved through standard read/write operations on memory-mapped registers with test-and-set semantics (hardware semaphores).

2.2 The SoC-TM Approach

Figure 2 provides a pictorial overview of our vertically integrated HW/SW support for transactional programming. At the heart of our proposal sits the Bloom module, which is in charge of snooping the addresses on the bus and managing conflicts on concurrent accesses to the shared memory region, and whose detailed functioning is described in Sec. 3. A small set of low-level primitives provides the necessary hooks to program the Bloom module from the software side. Through this API it is possible to define transaction boundaries, explicitly abort transactions or establish a maximum number of retries before switching to a serial mode of execution. To raise the level of abstraction at which final users will have to deal with transactional programming we extend the OpenMP programming model with directives to outline transactions in the application code. The task of lowering these abstractions into interactions with low-level services is left to the compiler, and to the runtime system. The basis for our software support is the OpenTM compiler [1], which we adapt and extend for our needs as described in Sec. 4.

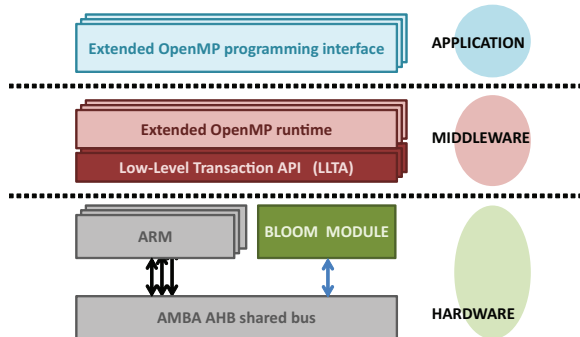


Figure 2: Overview of our vertically-integrated HW/SW transactional system.

3. HARDWARE TM DESIGN

In this Section we describe the details of our Transactional Memory Architecture. Our solution brings two key features. First, to meet the target requirement of low complexity, and to maximize design reusability, we devised a transactional memory architecture that does not involve any modifications to the hardware of the CPU. Neither custom instructions, nor changes to the pipeline are in fact needed to handle

transactional events. Our approach is based exclusively on software-triggered operations on memory mapped registers.

The second key feature is the clear separation of conflict detection from caches. Detecting data conflicts is one of the most critical and complex aspects of any transactional memory solution. For example, many hardware transactional memory systems require significant changes to the existing cache coherency protocol (e.g., [22]), which in turn put severe limitations to their applicability. Modifying the coherency protocol, in fact, affects both the timing and the testability of the design. For these reasons, we devised an external signature-based module that 1) monitors all the transactional accesses and saves them in per-core signatures, and 2) notifies the CPUs in case of data conflicts. We call this device the *Bloom module*. With this solution we are able to limit the modifications to the cache module to a few minor extensions, which do not involve the coherency protocol.

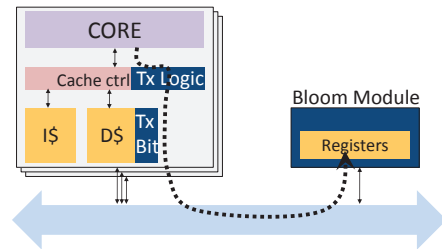


Figure 3: Transactional management logic (dark blocks represent added logic/registers).

Fig. 3 gives an overview of the proposed transactional architecture. The dark blocks represent the only hardware changes required by our solution. In particular, we can define three main components:

1. A new state bit (ie., the *Tx* bit) for each line of the Data Cache, which defines whether the data contained in the line is transactional or not.
2. Some new logic in the Cache Controller that handles the new transactional accesses.
3. The external *Bloom module*.

All the transactional events are triggered by regular read/write operations on memory mapped registers. For example, to start a transaction a core has to write to a particular register in the Bloom module. The cache controller detects that write operation, and sets an internal bit that enables the transactional logic.

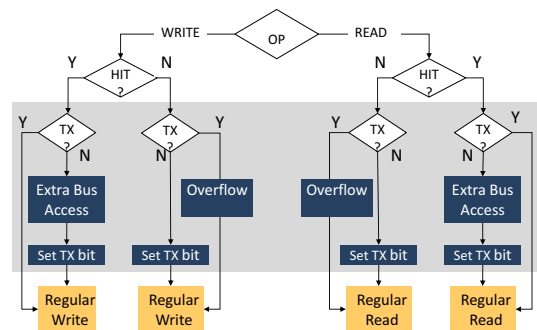


Figure 4: Modifications to the cache protocol for transactional reads and writes. The control flow within the gray block is new.

When executing transactional code, the transactional logic of the cache controller will eventually perform a few extra steps, as shown in Fig. 4. Notice that the bus protocol itself is not affected by these changes. Beside setting the Tx bit in the cache line, the cache controller must also manage two special cases: 1) carrying out an extra bus access (to notify the Bloom module) in case the line was already present in the cache before starting the transaction, and 2) performing a transaction-overflow when the line to replace is already transactional. The adopted protocol requires the overflowing core to notify the Bloom module by writing in one of its registers. The Bloom module will intercept that write, and set the requesting Core_ID in an overflow request queue (to deal with concurrent requests). The Bloom module will process the overflow request by resetting the entry from the request queue, and by sending an Overflow Interrupt Request (ie., Overflow-IRQ) to the other (non overflowing) cores. Interrupted cores enter an *idle* mode until, at the end of the overflowing transaction, the Bloom module sends a Wakeup-IRQ to restart their execution, with the overall effect of serializing transaction execution.

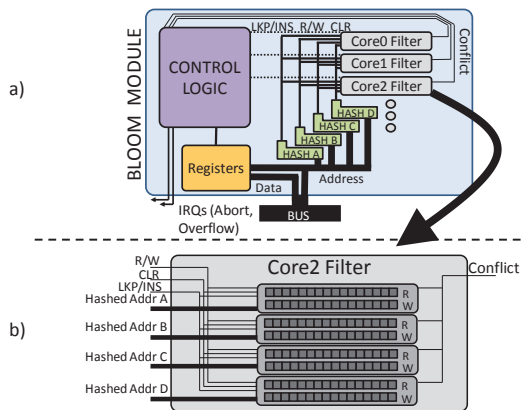


Figure 5: a) Overview of the Bloom module. b) Internal details of a Core Bloom Filter Unit (BFU)

3.1 The Bloom Module

As explained earlier, one of our key design goals was to decouple the transactional memory system from the cache coherency hardware. Signature-based transactional memory architectures have been proposed in the past [3, 24, 22] as an effective solution to disentangle conflict detection from caches. They all share the concept of adopting Bloom filters [2] to keep track of the read and write sets of transactions. Bloom filters have the property of potentially producing false positives when checking if a key has been entered into the set, but never false negatives.

Previous works have proposed per-core hardware signatures internal to each core. However, bringing the signatures inside the cores has some drawbacks. There are potential issues in terms of performance (e.g., when broadcasting the signatures at commit time [3]), complexity (e.g., modifications may be needed to the bus protocol [22]), and flexibility (e.g., no priority system, fixed conflict management [24]).

For such reasons, we developed the Bloom module as a single hardware device with access to snoop the shared bus. The physical proximity of the signatures facilitates conflict

detection and priority management, while the bus snoop approach uses the existing serialization of the shared bus provided by the bus arbiter. The internal details of the Bloom module are shown in Fig. 5a). The design consists of four main components: the control logic, the memory mapped registers, the Bloom Filter Units (ie., BFUs), and the hash functions. The control logic is responsible among other things for forcing the commit priority, and for coordinating the cores in case of special events (e.g., overflows, aborts). Note that the handshaking between the cores and Bloom module is entirely managed with commodity resources (e.g., interrupts and read/write memory operations), and no extra wires are required. Some of the functionalities of the Bloom module (e.g., commit priority) can be programmed at run time with simple writes on its memory mapped registers. The memory space reserved for the programming registers is quite small (only 256Bytes). As shown in Fig. 5b), each core has a BFU that consists of K read-write pairs of simple Bloom filters. We empirically found K=4 to give the best power/performance tradeoff, as we will show in Section 5. To limit the hardware requirements, we adopt a parallel Bloom filter design [21], which sets a single bit in K small bloom filters, instead of setting multiple bits in one large filter. Depending on the conflict resolution scheme, some transactions may need to be aborted in case of conflict. Note that while our target system lacks support for virtual memory¹, our approach could still be applied even on top of such an abstraction. As long as the shared memory space can be bounded within a precise address range, the OS can set the translated boundary addresses into two registers of the Bloom Module. Snooped addresses that fall within the translated range still identify shared data items.

Finally, we designed the hash functions with delay and power as major considerations. An ideal hash function should have small fan-out, small fan-in, and use few logic levels to keep the critical path short. It has been previously shown [25], that the lower order bits of an address are characterized by more randomness than the higher order bits. We thus decided to implement the hash function as a single level of two input XORing of lower order address bits.

3.2 Hardware Support for Ordered Commits

Auto-parallelizing compilers and directive-based parallel programming models (such as OpenMP) typically focus on regular data-intensive loops with no cross-iteration dependencies (DOALL). Speculative loop parallelization and thread-level speculation [26] [18] have been proposed in the past to effectively extract parallelism from programs whose execution cycles are mostly spent inside non DOALL loops or inside possibly inter-dependent tasks. In particular, some approaches based on TM and ordered commits have been proposed in the general purpose domain [12] [23]. Several embedded applications exhibit a similar pattern [8], thus making support for speculation very valuable in this domain. However, the cited TM-based works carry significant overheads, due to sophisticated software / hardware solutions. This makes them unsuitable for the embedded domain.

To enable speculative parallelization of target applications we enhance our Bloom module design with the capability of ensuring a commit order for transactions that respects the original sequential program order. Besides that, in gen-

¹our processors are MMU-less, which is not uncommon in the embedded domain.

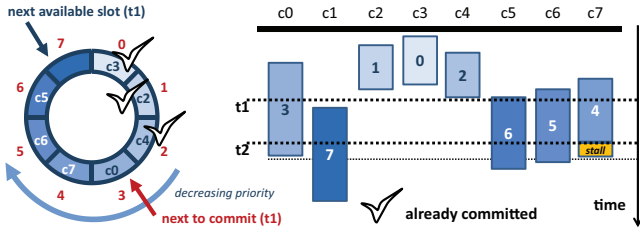


Figure 6: Circular buffer to support ordered commits of dynamically scheduled workload

eral to support speculative execution there also is the need for a mechanism to undo the effects on memory of transactions trying to commit ahead of time, when a real dependence arises. However, our HTM design uses a lazy-versioning, eager-conflict detection policy, which does not require to undo the effects of aborted transactions (all changes to shared data take place in the cache, and are only visible to shared memory upon commit).

Most parallel programming models provide facilities to schedule available workload among threads statically or dynamically. Parallel workload may be represented as a set of concurrent *tasks*. Statically assigning tasks to cores may result in better locality and improved predictability. Dynamically distributing tasks is preferred to solve load imbalancing issues. It is therefore important to support both schemes in *SoC-TM*, where each task is wrapped within a transaction.

With dynamic scheduling it is not possible to determine in advance which core will take ownership of a transaction. However, the scheduler itself will assign transactions to cores following the original program order. This order implies a notion of priority, in that a transaction T_i has higher priority than any transaction T_j for which $i < j$. Moreover, since the number of co-existing transactions in the system at any time² is bounded by the number of cores N , their priority can always be expressed as a number in the range $[0..N - 1]$ through a modulo operation, $Pri = TxID \% N$. To support ordered commits in case of dynamically scheduled workloads we consider a circular buffer of N slots, each associated to a specific priority level. We use two pointers to mark the “next available” slot in the buffer and the “next to commit” transaction. Every time that a core queries the system for a new transaction the physical ID of that core is inserted in the next available slot of the buffer, and the corresponding pointer is atomically shifted one slot ahead. This allows to keep at any instant in time a record of the order in which cores can commit their transactions. Upon reaching the commit point in a transaction every core accesses the Bloom module. The core whose ID is stored in the slot pointed by the “next to commit” pointer is allowed to complete its transaction, while the others are stalled. The pointer is eventually shifted to the next element in the buffer. In the example in Fig. 6 we consider $N = 8$. Color codes (shades of blue) are used to indicate priorities (the darker, the lower the priority). Slots of the circular buffer are thus labeled with transaction IDs whose priority level decrease clockwise. Due to dynamic workload scheduling, transaction dispatching is serialized and transactions are not tied to a specific core. At instant t_1 transactions 0-6 have been dispatched, and the “next available” pointer points to the sev-

²We consider a *run-to-completion* thread model

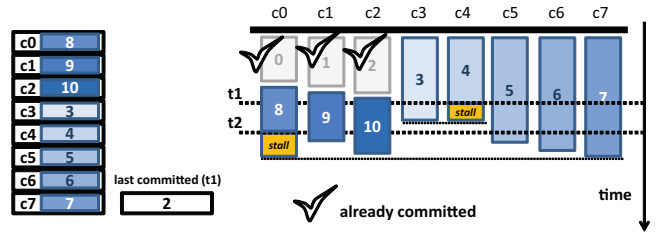


Figure 7: Priority buffers to support ordered commits of statically scheduled transactional workload

enth slot, where the ID of core 1 is to be annotated shortly afterwards. Transactions 0-2 have already committed, so the “next to commit” pointer points to slot 3. At instant t_2 core 7 will try to commit transaction 4, but the Bloom module will stall its execution until the point where core 0 will commit transaction 3.

With static scheduling the set of transactions that each core will be in charge of processing is fixed. Each thread computes locally the ID of the next transaction to be processed, based on the number of available cores and on its own physical ID. For this reason, to support ordered commits in this scenario we build a different buffering mechanism, with core-specific registers where the ID of the transaction being executed is stored. After a core computes the ID of the next iteration to process, it communicates this value to the Bloom module, which annotates it in the register associated with that core. A separate register is used to store the ID of the transaction that “last committed”. Upon arrival onto a commit point each transaction queries the Bloom module, which will only allow the transaction whose ID equals the content of the “last committed” register + 1 to commit, and stall the others. In Fig. 7 we show an example with $N = 8$. At instant t_1 transactions 0 to 2 have committed, and cores 0 to 2 have started executing transactions 8 to 10, whose IDs are stored in the respective registers. Core 4 tries to commit transaction 4, but the “last committed” register reads 2, so the core will be delayed until core $2+1=3$ commits. Similarly, when at instant t_2 core 0 tries to commit transaction 8 it is stalled until core 7 commits transaction 7.

4. SOFTWARE TM SUPPORT

Our Bloom module can be programmed via software by reading and writing into memory mapped registers. We developed a small set of low-level primitives which expose to the software a practical way of accomplishing this task. The functions of this transactional API are briefly described in Table 1. However, dealing with transactional behavior of the application at this very low level of abstraction would require extensive manual modification of the source code, which is a tedious and error-prone task, hindering the ease of use of such a programming abstraction. Higher level programming abstractions are desirable for the sake of simplifying embedded application development.

OpenMP is a well-recognized standard for shared memory parallelism, which provides an easy-to-use incremental approach to code parallelization based on compiler directives. OpenMP has been proposed several times as a convenient programming paradigm for MPSoCs as well [11] [17] [10]. Specific extensions to the standard programming interface have been proposed to efficiently leverage heterogeneous

Table 1: Low-level Transactional API (LLTA)

Function name	Brief description
<code>void begin_transaction ()</code>	Marks the beginning of a transaction by enabling conflict management on shared memory accesses
<code>void end_transaction ()</code>	Marks the end of a transaction by disabling conflict management on shared memory accesses
<code>void force_abort ()</code>	Forces the current transaction to abort
<code>int check_abort ()</code>	Returns 1 if the current transaction was aborted, 0 otherwise
<code>void bloom_max_retries (int n)</code>	Sets the maximum number of times a single transaction can be aborted before switching to serial mode
<code>void update_thread_priority (int pri)</code>	Sets current thread's commit priority
<code>void start_ordered_transaction ()</code>	Marks the beginning of an ordered transaction by enabling conflict management on shared memory accesses. Prevents current transaction from committing earlier than transactions with higher priority
<code>void end_ordered_transaction ()</code>	Marks the end of an ordered transaction by disabling conflict management on shared memory accesses and ordered commits

computing resources [17], accelerators [10] or memory hierarchies [11]. Similar to those approaches, specific extensions to OpenMP can be designed for transactional programming as well. In the general-purpose computing domain similar solutions have been proposed in the past few years, such as OpenTM from Stanford [1] or a similar proposal from the Barcelona Supercomputing Center [14]. OpenTM specifies a small set of extensions to the standard OpenMP 2.5 directives that enable programmers to wrap critical sections in their codes within transactions. There are three basic means to specify transactional workload in OpenTM. First, the boundaries of a transaction can be explicitly marked by enclosing a code region within a `transaction` directive.

```
#pragma omp transaction [clause[ [, ]clause]...]
    structured-block
```

Transactions can also be implicitly outlined by transactifying the workload generated by worksharing directives. A (group of) loop iterations can be enclosed within a transaction through the `transfor` directive:

```
#pragma omp transfor schedule (type, chunk)
for (i=LB; i<UB; i = i OP stride)
    structured-block
```

This construct allows to group *chunk* iterations of the target loop within a single transaction, scheduled according to the policy specified by *type* (static, dynamic, guided). Finally, task parallelism can also be transactified with the `transsections` directive:

```
#pragma omp transsections
[ #pragma omp transaction]
    structured-block-1
[ #pragma omp transaction]
    structured-block-2
...
```

A prototype implementation of the OpenTM directives was developed in the GCC 4.3.0 compiler, and the authors made the source code publicly available. The OpenTM compiler is capable of generating code which targets generic STM and HTM systems by emitting calls into custom runtime library functions which can be seen as generic primitives providing basic services for transactional programming. Their implementation can be specialized for a different target by leveraging platform-specific low-level facilities. Due to these considerations we decided to use the OpenTM programming interface and compiler as a starting point.

4.1 Compiler and Runtime Customization

Adapting the OpenTM framework to our MPSoC requires several modifications. First, the runtime library has been re-designed to cope with hardware and software limitations (memory space constraints, absence of MMU and OS support for virtual memory). For performance reasons, thread management does not rely on standard threading libraries, but directly controls hardware resources, which makes it extremely fast and lightweight. Barrier synchronization is also made very cheap by leveraging L1 SPMs, thus avoiding bus congestion due to busy waiting.

We also introduced a few modifications to the compiler to take advantage of the peculiarities of the memory subsystem of our MPSoC. One of the performance limiters of HTM systems is the difficulty of distinguishing between private and shared data [20, 25], which leads to increased coherence traffic and false sharing issues. The OpenMP programming model requires that all data items referenced within a `parallel` region are explicitly marked as `shared` or `private`.

```
#pragma omp parallel shared(a) private(i)
```

The compiler can thus leverage this information to automatically convey the allocation of private and shared variables onto distinct regions of the address space. Shared data is mapped onto the address segment which is under control of the Bloom module, whereas private data resides in those “private” regions in the L2 shared memory which are not managed transactionally. This disambiguates addresses and allows a lighter and faster conflict management.

Due to the cache-line granularity of our TM system, another issue which hinders performance is false sharing. If two different shared data items, accessed from within two different transactions, reside in the same cache line, an abort will be issued even if no real conflict takes place. This phenomenon can easily arise if data resides in addresses which are not aligned to cache line boundaries. Since extensive data alignment may lead to memory waste, rather than automatically and silently aligning all OpenMP `shared` objects, we provide a directive to control alignment of key data items. The custom `aligned` qualifier can be used instead of the `shared` clause to achieve this goal.

```
#pragma omp parallel aligned(a)
```

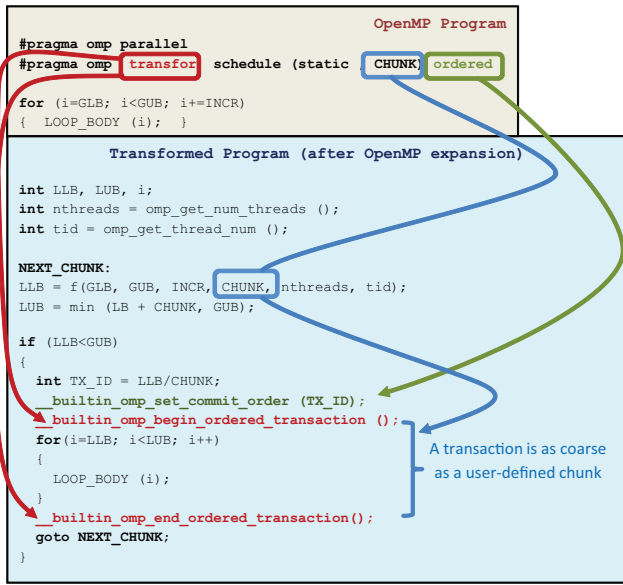


Figure 8: Speculative loop parallelization (static)

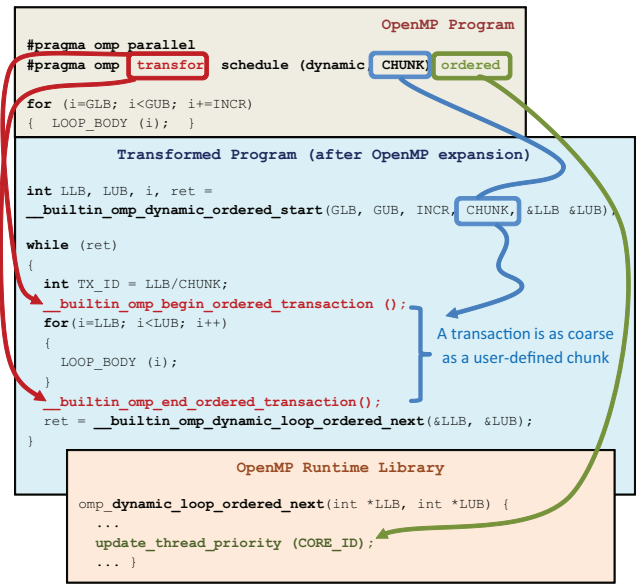


Figure 9: Speculative loop parallelization (dynamic)

4.2 Extensions for Ordered Commits

OpenMP provides an `ordered` construct which specifies a structured block in a loop region that will be executed in the order of the loop iterations. This sequentializes and orders the code within an ordered region while allowing code outside the region to run in parallel.

```
#pragma omp parallel for ordered
for (i=0; i<N; i++)
{ S1;
  S2;
  #pragma omp ordered
  { a[i+1] = a[i]; } }
```

In the code snippet above statements S1 and S2 from different loop iterations can be executed in any order, and thus are allowed to run in parallel. In contrast, the statement protected by the `#pragma omp ordered` construct must be executed following the sequential loop order. Similar loops where each iteration carries a data dependence with another iteration at a precise distance can be parallelized by synchronizing threads so that the dependent iteration (consumer) waits for its predecessor (producer) to update the shared data element before accessing it (DOACROSS). However, loops could carry dependencies that only manifest themselves rarely, while most iterations can be executed fully in parallel. In this case enclosing the loop body—or just the statement(s) possibly carrying the dependence—within a transaction and ensuring that transactions commit following the sequential program order ensures that the original loop semantics are preserved.

Authors of OpenTM suggest that the `ordered` clause in OpenMP can be coupled to transactional variants of work-sharing constructs (`transfor` and `transsections`). From the programming language point of view this would provide a convenient abstraction to enable speculative code parallelization. However, implementing a similar semantics requires the underlying TM system to be capable of ensuring transaction ordering, which is not discussed in [1].

As discussed in Sec. 3.2, our Bloom module has been designed with this ordering requirement in mind; the order in which transactions can commit can be dynamically programmed through the functions of the LLTA (see Tab. 1). In Figures 8 and 9 we show how to annotate a loop for speculative parallelization with static and dynamic scheduling, respectively, and how our compiler transforms it. We show in black the standard loop transformations made by the OpenMP compiler, and in red and green the modifications that we introduced to the `transfor` and `ordered` expansion code respectively. When a `CHUNK` size is provided to the `schedule` clause, OpenMP transforms loops in such a way that `CHUNK` consecutive iterations are executed within the inner loop (hereafter called the *chunk loop*) with lower and upper bounds (LLB, LUB) computed locally. If the schedule type is `static`, these bounds are computed based on the thread ID, the total number of threads and other parameters (e.g `CHUNK` size, loop stride, global lower and upper bounds). The chunk loop is wrapped within a transaction by marking its beginning and end with calls to the runtime functions `omp_begin_ordered_transaction` and `omp_end_ordered_transaction`. Within these functions the corresponding LLTA primitives are finally invoked. Before initiating a transaction, its global ID is computed within the thread (`TX_ID`), and is passed to the runtime function `omp_set_commit_order`, which notifies the Bloom module which core is currently executing the transaction by invoking the LLTA primitive `update_thread_priority`.

If the schedule type is `dynamic`, lower and upper bounds for the chunk loop are retrieved by invoking the iteration scheduler through calls to the runtime library. We have implemented custom extensions to these scheduling functions which augment the standard scheduler with the capability of programming the Bloom module to ensure that transactions commit in the program order. As shown in Fig. 9, our functions are called `omp_dynamic_loop_ordered_start` and `omp_dynamic_loop_ordered_next`, and from the latter the LLTA primitive `update_thread_priority` is invoked.

5. EXPERIMENTAL RESULTS

In this section we evaluate the proposed *SoC-TM* design. We tested our architecture with several hardware and software configurations. For convenience, we report the hardware parameters in Table 2.

Parameter	Configuration(s)
CPU	ARMv7, 3-stage in-order pipeline, 200Mhz
L1 cache	8KB 1-way Icache, 16KB 4-way Dcache
Cores	{1, 2, 4, 8}
Policies	Bloom-module, Locking
Scheduler	Software, Hardware
Signature	{2KBits 4-way} Read and Write filters

Table 2: Hardware configurations.

We compare two policies:

- **Locking.** The level of granularity (i.e., coarse vs. fine) depends on the specific application. Note that we employ a lightweight implementation of locks with simple read/write operations on hardware semaphores (no OS function calls, see Sec. 2.1);
- **Bloom-module.** Our *SoC-TM* architecture with the centralized Bloom module. The size of the Read and Write signatures contained in each BFU is 2Kbits.

We chose a representative subset of applications from the STAMP benchmark suite [15]. The workloads of these applications can be categorized according to their synchronization pattern and the length of their transactions: 1) large non-conflicting transactions (Vacation); 2) barrier-based synchronization with small transactions (Kmeans); 3) large transactions (Genome); mix of short and large transactions (Labyrinth). In our experiments we collected both the execution cycles and the energy of each component. For the Bloom module we only modeled the energy of the signatures, since they constitute the main source of energy consumption. Future work will also include the energy spent by the control logic and the hash functions. Fig. 10 shows the performance

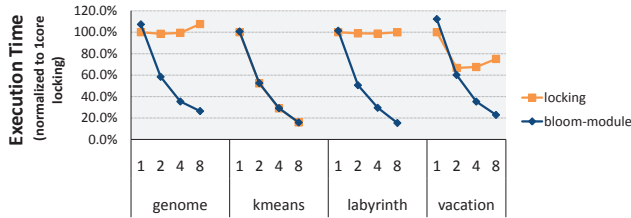


Figure 10: Performance scaling for Locking and Bloom-module configurations.

of the *SoC-TM* architecture, compared to a locking scheme. The values are normalized relative to the single core locking scheme. We see that for almost all benchmarks, *SoC-TM* achieves much better results than locking. The only exception is Kmeans, where *SoC-TM* performs the same as locking. This is expected, since in Kmeans the cores synchronize with barriers, and only 5% of the time are executing transactions. Also, *SoC-TM* is penalized with a 1core configuration, because of the extra bus accesses required to updated the Bloom module (as explained in Section 3).

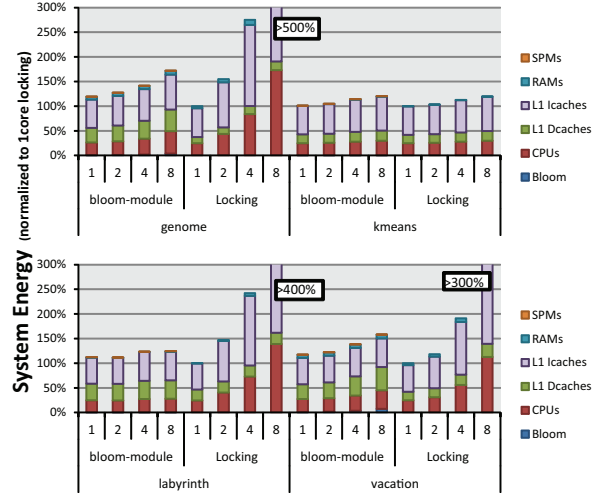


Figure 11: Energy breakdown for Locking and Bloom-module for STAMP benchmarks.

Fig. 11 shows the energy distributions for both locking and *SoC-TM*. All the values are normalized relative to the single core locking configuration. In general, *SoC-TM* gives better energy results than locking. Similar to performance results, *SoC-TM* is also penalized in terms of energy in a 1-core configuration by the extra bus accesses. Also, note that the higher the number of cores, the greater is the contribution of the Bloom module to the overall energy consumption. This is expected, since the number of hashing operations grows linearly with the number of cores. In the worst case (i.e., with 8 cores) the Bloom module consumes about 5% of the system energy, as shown in Fig. 12.

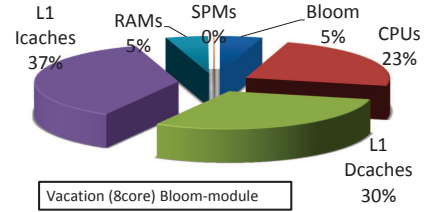


Figure 12: Energy Distribution for Vacation using the Bloom-module configuration with 8 cores.

5.1 Speculative parallelization

In this section we describe our experiments with speculative parallelization of applications. We first consider a synthetic benchmark designed to explore the behavior of our *SoC-TM* when ordered commits are imposed on a loop carrying cross-iteration dependencies. This benchmark consists of a loop containing a certain amount of fully parallel “dummy” work (statement S1)—which can be executed in any order by any thread—plus an instruction (statement S2) carrying a forward dependence (S1 is independent of S2).

```

for (i=0; i<WORK; i++) {
S1: DUMMY_WORK (w);
S2: a[i] = a[i + dep_dist]; }

```

To model varying amounts of independent work we consider different values of a parameter $w = \{0, 10, 100\}$, passed as an argument to the function `DUMMY_WORK`, called in statement S1. $w = 0$ indicates that the loop only contains the instruction creating the dependence (S2), $w = N (> 0)$ indicates that the independent work in the loop accounts for (roughly) N times the execution time spent in dependent computation (i.e., $time(S1)/time(S2) = N$).

The cross-iteration dependence has a parameterized distance $dep_dist = \{0, 8\}$. We consider $\{dep_dist = 0, w = 0\}$ to be a worst case for our system, since there is almost no parallelism among instructions; each iteration depends on the previous, and there is no independent work in the loop which can be executed in parallel. For larger values of w we expect to be able to extract more and more parallelism (DOACROSS). Intuitively, when dep_dist is bigger than the number of cores it can not happen that two threads holding inter-dependent iterations are executing concurrently, and thus the loop should execute fully parallel (DOALL). We thus also consider $dep_dist = 8$ (where 8 is the maximum number of cores in our system).

We parallelize the loop with dynamic scheduling. As shown in Fig. 9, when this scheduling type is employed the OpenMP compiler generates calls to the runtime scheduler to retrieve lower and upper bounds of a loop which spans `CHUNK` iterations. The scheduler holds a global iteration counter. At each invocation from any core the iteration counter is locked. The scheduler atomically returns to the requesting core the next available iteration, and increments the counter by the chunk size. This scheduling technique carries a significant overhead when `CHUNK` is very small (e.g., 1) and the amount of work within a loop iteration is very small. Increasing the size of `CHUNK` helps in hiding the overhead, since the scheduler is invoked less frequently. However, assigning multiple consecutive iterations to threads may break dependences (if the dependence distance is smaller than the `CHUNK` size), thus resulting in an increased abort rate. To reduce the cost for dynamic scheduling we accelerate it in hardware, extending the Bloom module and the LLTA.

The results for this experiment are shown in Fig. 13, where we compare the scaling of dynamic speculative loop parallelization using both software and hardware iteration scheduling, and considering aligned and unaligned data. The first thing to notice is that hardware iteration scheduling is extremely beneficial when the loop body contains small amounts of work. As expected, when $\{dep_dist = 0, w = 0\}$ the system only scales up to 2 cores, then there is not enough parallelism and frequent aborts take place, due to dependence violations. Clearly this effect is mitigated when w increases. For $w = 10$ the system scales up to 4 cores, and with $w = 100$ it perfectly scales up to 8 cores. When $dep_dist = 8$, as expected, the aborts due to cross-iteration dependence violation almost completely disappear. However we still see aborts due to false conflicts implied by bad data alignment. Our `aligned` clause allows much better scaling.

Besides this synthetic benchmark we evaluate our support to speculative parallelization with six real benchmarks, extracted from two representative benchmark suites for embedded systems: MiBench [6] and EEMBC [4]. The target applications were selected because most of their execution time is spent inside non DOALL loops [8] (data-intensive or functional loops). Our approach to parallelizing these applications was simply that of annotating these loops with

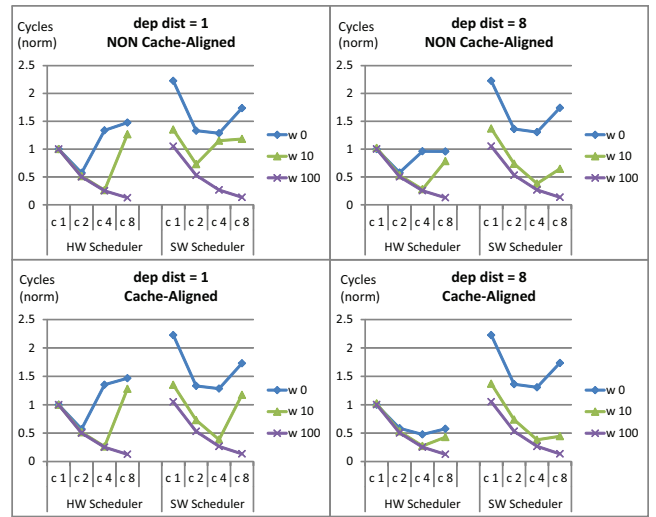


Figure 13: Execution time scaling (synthetic).

our custom directives for ordered commits. Speedup scaling results are shown in Fig. 14, considering HW and SW schedulers and aligned and unaligned data. For programs

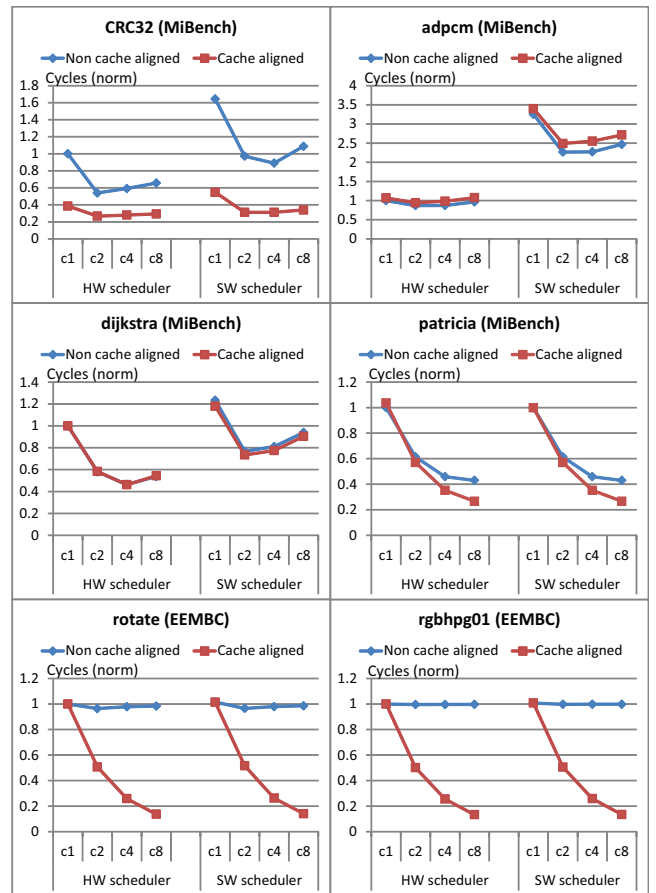


Figure 14: Execution time scaling (real programs).

`adpcm` and `CRC32` each transaction executes very few or no instructions besides the one(s) carrying the dependence. Thus,

similar to what we noticed for the synthetic benchmark with “dummy work” $w = 0$ there is no parallelism to be extracted among threads. In this case it is important to point out that our system has a very lightweight management of conflicts, as the execution time does not increase with an increasing number of processors even if the degree of contention does. For these two benchmarks and for `dijkstra` it is also possible to notice a significant benefit introduced by the use of the hardware scheduler. Indeed, due to the small amount of work within each transaction, the cost for invoking the software scheduler has a non-negligible impact on performance.

The `patricia` benchmark carries a different dependence pattern, in that the dependence may manifest or not depending on the input set. This causes rare dependence violations at runtime, which allow an almost perfect scaling. It is also possible to notice that if data is not cache-aligned a higher number of aborts happens, due to false conflicts.

Finally, programs `rotate` and `rgbhpg01` behave as DO-ALL loops once false conflicts due to poor data alignment are removed with our `aligned` clause, which leads to a perfect scaling. The difference between SW and HW instruction scheduling is completely negligible, since transactions contain amounts of work large enough to amortize the overhead. Overall, the memory waste due to shared data alignment is very small for all benchmarks (in the worst-alignment case: 58 Bytes on average, up to 132 Bytes for `adpcm`).

6. CONCLUSION AND FUTURE WORK

We have presented *SoC-TM*, a vertically integrated HW-SW framework for transactional programming on embedded MPSoCs. Easing application development is of the utmost importance in this domain, where programmers are responsible for fully harnessing the compute potential of multi-cores. *SoC-TM* eases application development on shared memory MPSoCs leveraging a lightweight yet efficient HTM design, coupled with the simplicity of directive-based programming of OpenMP. Speculative parallelization is also supported, which makes the task of extracting parallelism from sequential codes less cumbersome. Our results confirm the effectiveness of *SoC-TM*, concerning ease of use, performance, simplicity of design and energy consumption. We are planning to extend our work in several directions. First, we plan a more extensive evaluation of our framework, comparing against other state-of-the-art TM systems and considering more complex scenarios where multiple tasks may be time-multiplexed on the same core by the OS. Second, compiler analysis can be leveraged to automatically identify loops amenable to speculative parallelization by estimating the independent work within each transaction. Finally, we plan to implement our programming model based on the latest (3.0) OpenMP specification, where the `task` construct enables very flexible support for fine-grained TLS.

7. REFERENCES

- [1] W. Baek, C. C. Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun. The OpenTM transactional application programming interface. In *PACT*, pages 376–387, 2007.
- [2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, July 1970.
- [3] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA*, pages 227–238, 2006.
- [4] EEMBC. Eembc, the embedded microprocessor benchmark consortium. <http://www.eembc.org>.
- [5] C. Ferri, S. Wood, T. Moreshet, R. I. Bahar, and M. Herlihy. Embedded-tm: Energy and complexity-effective hardware transactional memory for embedded multicore systems. *Journal of Parallel and Distributed Computing*, 70(10):1042–1052, October 2010.
- [6] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on Workload Characterization*, pages 3–14, 2001.
- [7] T. Harris, J. R. Larus, and R. Rajwar. Transactional memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.
- [8] A. Kejariwal, A. V. Veidenbaum, A. Nicolau, M. Girkar, X. Tian, and H. Saito. On the exploitation of loop-level parallelism in embedded applications. *ACM Trans. Embed. Comput. Syst.*, 8:10:1–10:34, February 2009.
- [9] L. Kunz, G. Girão, and F. Wagner. Evaluation of a hardware transactional memory model in an NoC-based embedded MPSoC. In *SBCCI*, pages 85–90, São Paulo, Brazil, 2010.
- [10] F. Liu and V. Chaudhary. Extending OpenMP for heterogeneous chip multiprocessors. In *International Conference on Parallel Processing*, pages 161–168, 2003.
- [11] A. Marongiu and L. Benini. An OpenMP compiler for efficient use of distributed scratchpad memory in MPSoCs. *Computers, IEEE Transactions on*, PP(99):1, 2010.
- [12] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. *SIGPLAN Not.*, 44:166–176, June 2009.
- [13] Q. Meunier and F. Petrot. Lightweight transactional memory systems for nocs based architectures: Design, implementation and comparison of two policies. *Journal of Parallel and Distributed Computing*, 70(10):1024–1041, October 2010.
- [14] M. Milovanovic, R. Ferrer, O. Unsal, A. Cristal, X. Martorell, E. Ayguadé, J. Labarta, and M. Valero. Transactional memory and OpenMP. In B. Chapman, editor, *A Practical Programming Model for the Multi-Core Era*, pages 37–53. Springer Berlin/Heidelberg, 2008.
- [15] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *International Symposium on Workload Characterization*, Sept. 2008.
- [16] NVIDIA website. NVIDIA Tegra-2. <http://www.nvidia.com/object/tegra-2.html>.
- [17] K. O’Brien, K. O’Brien, Z. Sura, T. Chen, and T. Zhang. Supporting OpenMP on cell. In *International Workshop on OpenMP*, pages 65–76, 2008.
- [18] M. K. Prabhu and K. Olukotun. Exposing speculative thread parallelism in spec2000. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 142–152, 2005.
- [19] Qualcomm Inc. Snapdragon MSM8660 and APQ8060 Product Brief. <http://www.qualcomm.com/documents/snapdragon-msm8x60-apq8060-product-brief>.
- [20] R. Quisilant, E. Gutierrez, O. Plata, and E. L. Zapata. Improving signatures by locality exploitation for transactional memory. In *PACT*, pages 303–312, 2009.
- [21] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *International Symposium on Microarchitecture*, pages 123–133, 2007.
- [22] A. Shriraman, S. Dwarkadas, and M. L. Scott. Implementation tradeoffs in the design of flexible transactional memory support. *Journal of Parallel and Distributed Computing*, 70(10):1068–1084, October 2010.
- [23] C. von Praun, L. Ceze, and C. Cascaval. Implicit parallelism with ordered transactions. In *12th symposium on Principles and practice of parallel programming, PPOPP ’07*, pages 79–89, New York, NY, USA, 2007. ACM.
- [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 *HPCA*, pages 261–272, 2007.
- [25] L. Yen, S. C. Draper, and M. D. Hill. Notary: Hardware techniques to enhance signatures. In *International Symposium on Microarchitecture*, pages 234–245, 2008.
- [26] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *HPCA*, pages 290–301, feb. 2008.