# Second Year PhD Report
# 2011/2012

Alessandro Pellegrini
pellegrini@dis.uniroma1.it
DIAG - Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Sapienza, University of Rome
Via Ariosto, 25 - 00185 - Rome - Italy

October 3, 2012

*This document is meant to give information concerning the research activities related to the first two years of the PhD Program in Computer Science. In particular, after pointing out the Research Context I am focusing on, and its current State of the Art, I will concentrate on the results which I have so far achieved—showing their relations to the addressed problems—and will provide a description of the Future Work which I intend to follow through during my last year.*

## 1 Research Context

The total number of transistors available on a microchip, over the past 45 years, has doubled every 18–24 months, a trend which is known as Moore's law [78]. This electronic advancement, due to improvements in production cycles and technologies, had a direct impact on computation speed of single processors—and therefore on the actual efficiency of algorithms and programs run onto them. This computation power's enhancement trend has not significantly changed until year 2003 [99], when physical constraints related to power consuption [60] have produced a final stall in the growth of per-CPU clock frequency.

Nevertheless, both industrial and academic institutions are still demanding for an increased computation power, which is the basis for the development of new/more efficient programs and algorithms, and for addressing always more complex problems. This has lead the industry to the development of new computing architectures which provide multiple processing units, and therefore give the possibility to enhance software performance by relying on the parallel programming paradigm. Among the various proposals, the most promising ones are GPUs (see, e.g. [46, 80, 36, 73]) and the multi/many-core architectures (see, e.g., [22, 25, 82, 93]). While the former is more targeted at data-parallel applications [107], the latter is (generally speaking) a more widely applicable paradigm, and it is indeed the one which I have targeted for my research activity.

Today, parallel programming is indeed one of the possible answers to the physical limits in the development of more powerful computing systems. If this kind of system has reached a sort of stability from the electronic point of view (i.e., the format of architectures is now stabilized), there's still a long way before the software run on top of it will be considered both efficient and widely exploitable. In fact, the ability to produce effective and efficient parallel code has been so far the privilege of few scientist, and therefore considered a niche field.

The need for parallel processing is evident in real-time computing, scientific and non-scientific areas. Scientific computing in high-level languages requires high-fidelity simulations and computations that can be achieved by increasing the number of parameters and the size of the datasets. Similarly, in the field of embedded real-time computing the replacement of analog receiver technology (in sensor arrays) by digital technology requires faster digital processing capabilities at the sensor front end. Additionally, the migration of more and more post-processing—such as tracking and target recognition—to the sensor front end necessitates increasingly powerful processing architectures. Finally, as said, multicore processing units are now a consolidated reality on out-of-the-shelf machines, giving normal users a relevant computation power which is most of the times wasted or not fully exploited.

My research is precisely focusing on this direction, that is, the possibility to produce tools and techniques which might enhance the effectiveness of parallel programming and bring its power directly to inexperienced programmers, or scientists from different fields, which do not have the skills required to fully exploit the always increasing computation power offered by multi/many-cores architectures, and yet can be considered amongst the ones which could benefit the most from it.

As hinted, my research is mostly twofold. On the one side, I am focusing on *performance* aspects of concurrent programming, in particular exploring the viability of new synchronization patterns and protocols (e.g., non-blocking algorithms, which are algorithms ensuring that threads competing for a shared resource do not have their execution indefinitely postponed by mutual exclusion).

On the other side, I am concentrating on *transparency*, which can be regarded as the answer to the demand for providing software developers with efficient tools and techniques for supporting the creation of parallel algorithms/programs. In order to provide the user with full transparency, I have to develop tools which are able to understand at their best what the programmer is really asking the machine to do, that is, tools which are able to understand both the program and the machine which the program will be run on. In addition, to make these tools general purpose, a low-level approach should be used: Every program talks with the underlying hardware using machine code, therefore producing tools and methodologies which are able to directly handle final executable files will produce a benefit for programmers of multiple languages. This means that if we want to produce a general-purpose tool, we cannot move from a language of a level higher than machine code, or its 1:1 companion, the assembly.

So far, I have concentrated my effort on some special cases, i.e. Parallel Discrete Event Simulation (PDES) systems [43] and Transactional Memories [54, 94, 74, 31]. The first one falls into the Event-Driven Programming [72] paradigm, in which the flow of the program is determined by events (generated from the users or from other software modules) or messages from other programs or threads. PDES, in particular, is used to implement simulation models through a reduced-size set of APIs offered by simulation platforms. Event scheduling can be based on the *safety* property of events (i.e., if an event will not cause any time-causality inconsistency), or using an optimistic approach [61], where events are executed regardless of their safety (thus being intrinsically prone to great exploitation of parallelism), and if an inconsistency is later found, a rollback operation (supported by state log/restore operations) is performed, bringing the state back to consistency.

On the other hand, Transactional Memories are a generic non-blocking synchronization construct, which have been studied for over a decade and offer—in their software version—both obstruction-free [53, 49] and lock-free [94, 41] implementations. They allow a correct sequential object to be mapped automatically into a correct concurrent object, using the definition of a *transaction*, which is a sequence of instructions which atomically modifies a set of data objects. To fully exploit transactions, the user must explicitly mark in the code which regions have to be executed atomically. Later on, at runtime, the underlying framework uses a system-wide declaration of a transaction's update intention, to notify any other concurrently-running transaction that an update to a particular object is about to be performed.

# 2 State of the Art

In the context of performance, reducing the impact of mutual exclusion has been considered a benefit since the early 1970's [34]. Lamport [67] gave the first non-blocking algorithm for the problem of a single-writer/multiple-reader shared variable. Herlihy [52] proved that for non-blocking implementations of most interesting data types (e.g., linked lists), a synchronization primitive that is universal, in conjunction with reads and writes, is both necessary and sufficient. A universal primitive is one that can solve the consensus problem [38] for any number of processes. Among various ones, atomic operations like *compare and swap* (`CAS`) and *load-linked/store-conditional* (`LL/SC`) [27, 62, 55, 52] can be exploited as universal primitives, avoiding costly ones like *spinlocks*, and being safe from drawbacks like *deadlocks*, *priority inversions*, and *convoying*.

A subtle problem associated with most lock-free algorithms is the ABA problem. It was first reported in association with the introduction of the `CAS` instruction on the IBM System 370 [27]. It occurs when a thread T1 reads a value A from a shared object and then an interrupting thread T2 modifies the value of the shared object from A to B and then back to A. When T1 resumes, it erroneously assumes that the object has not been modified. Given such behavior, there is a serious risk that T2's execution is going to violate the correctness of the object's semantic. Practical solutions to the ABA problem include the use of hazard pointers [76] or the association of a version counter to each element in platforms supporting a double-word compare-and-swap primitive (CAS2) such as IA-32 [58, 59].

In literature, a relevant amount of researchers has addressed the study of non-blocking algorithms from a data-structure point of view, i.e. several data types along with non-blocking procedures to handle them have been proposed. The works in [64, 77, 53, 48] address, e.g., the implementation of queues and lists.

Concrete applications of non-blocking algorithms appear in solutions for the mutual exclusion problem [23, 68, 106], write barrier implementations in garbage collectors [71], and in implementations of composite locks [51]. All these solutions have been experimented in real implementations of programs, proving the effectiveness and viability of the approach.

Recently, the work in [65] has proposed a general methodology, namely *fast-path-slow-path*, the idea of which is to build data structures from fast and slow paths, where the former ensures good performance, while the latter serves as a fall-back to achieve wait-freedom. Normally, the fast path is a customized version of a lock-free algorithm, while the slow path is a customized version of a wait-free one. A thread makes several attempts to apply an operation on the fast path; only if it fails to complete, it switches to the slow path, where the completion is guaranteed.

As mentioned before, Software Transactional Memories (STM) [45] are an answer to the high contention derived by the explicit use of locks when a high degree of parallelism is shown, expecially on a reduced set of data structures, which is likely to happen in the context of, e.g., high performance computing. The basic principle behind them is rather easy, involving the programmer in the conversion of critical sections into transactions, i.e. portions of code which should be executed atomically. Rather than that, the execution is non-atomic, but works on actual copies of the data, which are a-posteriori reconciliated with the main (global) version through a commit phase. In particular, during the commit phase, inconsistent accesses by different threads are detected, and transactions are therefore aborted and restarted, in order to take into account the new changes in data by other threads. Indeed, a transaction is a very intuitive abstraction that has been used with success in databases for a long time. Nevertheless, it is not a fully transparent solution[1], given that the programmer must once more determine which are the atomic portions of its code and explicitly mark them, using the API provided by the underlying library.

The idea underlying transactional memories is more widely applied by Speculative Processing, which can be regarded as a means to improve parallel programs' performance with respect to serial fractions. In particular, it tries to guess which is going to be the outcome of parallel fractions of programs and tries to execute the serial ones concurrently. By using a retry-until-commit approach, Speculative Programming does some work, the result of which may be incorrect, but if it is not, a significant increase in performance can be obtained.

This technique has been successfully applied in a number of different fields such as pipelined computing architectures [66, 92] and high performance computing systems and applications [28, 95, 91, 88]. Using Speculative Programming, parallel parts of the application are not subject to locks, therefore the processing units are fully exploited, and with some probability the partial results will be committed, with some other probability they will be undone. Overall, the global performance of the application will benefit from this approach.

The work in [24] has targeted non-replicated real-time databases and shows the benefits, in terms of transaction timeliness, by speculatively forking, upon detection of a conflict, a copy of the current transaction that remains idle and serves as a save-point to reduce the rollback cost.

Self-Adjusting Computation, in which programs respond to input changes by updating automatically their output [19, 21, 20, 47], is a technique allowing, to some extent, to advance in a computation—which is not necessarily consistent—until new information is available (i.e., updates on constrained data are performed) and the global execution is corrected using it. This is done by recording data and control dependencies during a program's execution, and by exploiting a propagation algorithm which updates the computation as if the program were run from scratch. To increase performance, propagation algorithms are able to perform these updates in an incremental fashion, i.e. only parts of the program which are affected by the data changes are re-executed.

Reactive languages are imperative languages which support the dataflow model of computation [44, 70, 47, 79]. They are mostly (but not necessarily) visual languages, and allow the user to manage the flow graph by putting links between various entities using some sort of GUIs. They mainly focus on how the different data components in the program connect, so that the procedures updating them are just a secondary aspect with respect to the actual graph defining the information *flowing* through the execution. In this way, given that the constraints on data are well-specified, some sort of runtime library is able to guarantee consistency on the dataflow whenever the data is changed by a portion of the software.

Both self-adjusting computation and reactive languages allow the end user to provide efficient execution of

[1] Nevertheless, STMs are definitely oriented to transparency, since synchronization is ensured and guaranteed transparently by the underlying library.

their parallel algorithms, although they are far from transparent. In fact, the programmer is required to explicitly know that he is developing software which will be run on concurrent machines, and some effort must be put in consistently defining which are the relations about data and procedures updating them, in order to allow the underlying runtime library to effectively maintain consistency, whenever some constraint is violated.

The work in [30] has proposed an extension to the C/C++ languages where the programmer is able to mark a memory region (i.e., either complex data structures, or even single primitive datatypes) as *reactive*. The executable, at compile time, is statically analyzed in order to identify which operations can produce modifications on data. In particular, every memory region which is explicitly marked as being reactive (through the exploitation of a modified version of the `malloc` allocator) is placed into protected pages, so that whenever they are accessed, a `SIGSEGV` signal is raised, and a specific routine can generate dependencies and reconciliate constraints.

PDES systems [43] are usually regarded as an effective means to carry on complex simulations, as an instance of the more general event-driven programming approach, by relying on multiple processing units either in a multi/many-core machine, or in a cluster. Many implementation provide scheduling mechanisms implemented using global data structures like *global queues* (see, e.g., [75], or allow particular supporting routines (e.g., communication) to be executed on separate threads (see again, e.g., [75]), imposing scalability constraints on the number of concurrent instances of simulation kernels being run.

In the optimistic simulation context [61], several solutions have been introduced for logging the whole state of a simulation object (at each event execution or after an interval of executed events) [39, 84, 87, 89], or incrementally logging modified state portions [90, 98, 105], or supporting a mix of the two approaches [40, 96]. With these solutions there is the need (i) to supply the necessary code to collect snapshots of the objects' state inside the application level software, or (ii) to employ calls to functions within the API of proper checkpointing libraries, or (iii) to statically identify (e.g., at compile-time) which portions of the address space need to be considered part of the state. Consequently, perfect transparency is not supported since the programmer must necessarily be faced with issues related to state snapshots, and hence parallelism and synchronization. Also, static identification of the memory locations to be included inside the snapshot is non-compatible with dynamic memory allocation/deallocation (e.g. via standard libraries) at the simulation object level. This is the case for the work in [105], which has some technical similarities to my work on the side of automatic instrumentation, but does not allow dynamic memory to be employed, thus not supporting recoverability for each permitted operation (allocation, deallocation and update).

The issue of dynamic memory based states for optimistic simulation objects has also been addressed by the optimistic simulation framework in [97]. However, ad-hoc APIs are used to explicitly notify to the simulation kernel that specific allocation/deallocation operations, and, more in general, operations on data structures based on dynamic memory (e.g. lists), need to be rollbackable. Hence, differently from my approach, dynamic memory based layouts via ANSI-C memory allocation/deallocation services are not supported.

# 3 Achieved Results

As mentioned earlier in this report, my research activity is focusing on two sides of the same coin, namely methodologies and techniques for bridging the end user with both parallelism transparency and performance on multi/many-cores architectures. In this section I provide a discussion of the so-far achieved results. In the beginning, I will present a technological result—showing a tool which stands itself as a building block for later works—emphasizing its capabilities, along with examples of application scope. Then, I will present methodological and design/implementation results related to the context ofEvent-Driven Programming and Software Transactional Memories.

## 3.1 Software Manipulation Framework

The work in [10] has paved the way to the possibility of altering (at compile time) the actual operations performed by an executable without modifying its semantics, i.e. the program's outcome is left unmodified. A tool implementing this technique, called *static software instrumentation*, has been realized via a software Parser/Modifier (PM) specifically designed for analyzing and rewriting ELF (Executable and Linkable Format) objects generated by standard `gcc` compilers (versions 3 and 4) for IA-32 and x86-64 architectures[2]. At the very base, PM works by parsing the object generated after linking together all the application level modules, and build in memory an

[2]At the time of this writing, an extension of the described tool/methodology is currently under active development, in order to support different architectures and executables' formats.

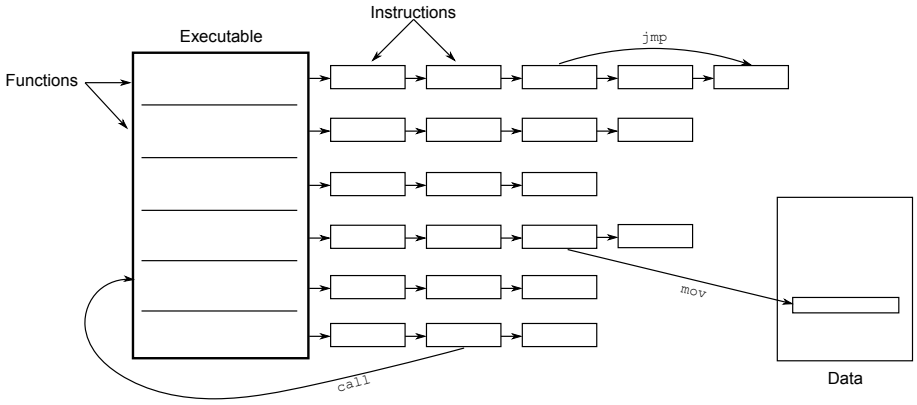



Figure 1: PM Internal Representation of Executables

intermediate representation which describes procedures and data structures (i.e., memory addresses) touched during the execution. In particular, every instruction in the program being analyzed is classified into a particular family, describing its actual behaviour (in terms of memory access patterns, flow control, ...). The internal representation is depicted in Figure 1, where some instruction/instruction and instruction/data dependencies are shown as well. The instrumentation process basically involves re-ordering or inserting new elements between the ones representing instructions, or changing depndencies pointers between instructions.

The user relying on PM is therefore allowed to specify instrumentation rules (specified in an xml configuration file), i.e. he is able to replace or prepend/append particular user-defined calls to specific instructions (depending on their family), analyze or modify particular functions, injecting new code into the executable, and/or produce multiple versions of the same original executable (with different instrumented behaviours) into the same program. PM simply applies rules to the intermediate representation of the instructions being analyzed, and in the end it creates a new executable, which transparently embodies all the changes.

Among the various benefits which can derive from the usage of this tool, I mention:

- *Profiling*: Code profiling is undoubtfully a useful technique to measure the efficiency of algorithms' implementations. Traditionally, in order to profile code, the user is required to either (i) manually modify the sources adding timers or use automatic source level instrumentation (therefore producing different versions of the code), or (ii) to rely on features provided by compilers, like the ones provided by the Intel Compiler [29] which is not necessarily available in any production cycle or might be incompatible with specific extensions required by the sources, or (iii) to use external tools which are targeted at fixed-size instruction sets, and therefore not widely applicable on out-of-the-shelf machines (see, e.g., [35, 85]) or (iv) are quite intrusive since they rely on dynamic instrumentation (see, e.g., [42, 33, 83, 32, 17]), or require the user to rely on specific APIs (see, e.g., [69]), or (v) rely on emulation rather than execution (see, e.g., [103]).

  On the other hand, relying on a tool like the one I have proposed, user can just maintain one single version of the sources, and write rules to inject efficient profiling routines to track the execution performance of single functions, class of functions or even single code snippets.

- *Performance Enhancements*: In order to enhance the performance of software architectures/algorithms, a viable approach is to rely on autonomic computing [50, 56, 63]. This technique allows a software platform to self-adjust its internal configuration in response to variation of the dynamic workload during its actual execution, in a way transparent to the user, i.e. the system platform is able to sense the environment and apply changes to itself in order to maximize some internal metrics. In [8], as explained in Section 3.2.2, I have presented an approach for supporting autonomic computing in the context of PDES simulations which exploits PM for creating multiple differently-instrumented versions of the same application-level code, activating the execution of either of the two by simply rewriting function pointers, depending on the current execution dynamics.

- *Synchronization Transparency*: having the software run on top of a parallel environment means that we are explicitly trying to have it run efficiently, in order to optimize in some way the overall performance. In this context, a great importance falls on the primitives which are actually used in order to enforce synchronization. In particular, in several contexts some primitives might be proven to be more efficient than

others, but in general this would be not an axiom (e.g., spinlocks are efficients when dealing with a small number of threads, while in other execution scenarios condition variables might provide a less intrusive synchronization form). PM offers the possibility to alter the actual behaviour of the program being developed for execution on parallel architectures in order to insert control routines aimed at fine tuning the running synchronization scheme as a function of the actual scale of the system (which can perfectly target dynamically changing systems), the number of active threads, and any other deploy parameter which can strongly affect the overall performance because of enhanced contention on the synchronization primitives.

- *Post-Mortem Debugging* [81]: The problem of finding bugs in a program is non-trivial, especially when in complex systems it is not possibile to reproduce the problem in the development/testing phase. On the contrary, if the production version of the software is lightweightly instrumented, it is possible to trace execution and store light metadata which allow, by analyzing a core dump, to step back between instructions (restoring previous snapshots of the memory map) and find where the actual bug is. PM actually provides all the facilities needed to support this kind of system, if ad-hoc routines are injected into the executable.

A subtle technical problem which might arise in instrumentation is related to how common compilers translate high-level languages structures in assemly. In particular, the `switch/case` construct, which is indeed very relevant in the context of event-driven programming, is translated using *indirect branches*, as specified by the System V ABI [100, 101, 18].

The modification of the actual executable leads to a resize of the sections associated with the object file, and to the shift of instructions and other memory locations inside the object layout. Given that instructions/data instructions/functions references are explicitly handled in the intermediate representation by PM, when the new executable is flushed, the headers associated with the ELF object, the relocation tables, and the offsets used for the identification of memory addresses referenced by the software—e.g. the destination addresses for `jmp` instructions—are rebuilt from scratch, in order to maintain references' consistency.

This is not the case for the aforementioned *indirect branches*, where the destination address is dynamically identified via the content of CPU registers. To cope with this issue, if such instructions are found in the executable being parsed, PM automatically flags the generation of a runtime monitoring module for supporting on-the-fly correction of destination addresses in register jumps. This mechanism is based on the insertion of a `call` instruction to an assembly-level monitoring module, referred to as `branch_corrector`, prior to each register jump in the original software. This monitoring module relies on a hash table where entries are structured as follows:

```
struct entry {
    unsigned long insn_addr;
    char flags;
    char base;
    char index;
    char scale;
    long displacement;
};
```

where `insn_addr` is the virtual address associated with the indirect branch instruction and acts as the key value for this table which is scanned by the `branch_corrector` module using a fast binary search. In particular, it reads from the stack its return value—which is actually the address of the subsequent instruction, i.e. the indirect branch—and associates a particular invocation of the monitor with the instruction which generated the insertion of the actual call.

This table is built and populated at compile-time during the instrumentation process—to avoid costly run-time disassembling techniques—and allows the `branch_corrector` module to fastly compute the target address of the jump, according to the IA-32/x86-64 specification (for a complete discussion on this, see [57]). To provide a lightweight mechanism for address correction, PM generates a second table at compile-time, which is visible only to `branch_corrector`. Each entry inside this table identifies an interval of addresses for which the instrumentation process gave rise to the same amount of shift inside the final (instrumented) memory layout. Such an offset is also maintained in the table entry. The table is ordered by interval extremes, and `branch_corrector` performs a logarithmic-cost binary search to retrieve the interval containing the original destination for the register jump, and the offset to be applied for the correction. Such a correction cannot however be applied by modifying the values of the CPU registers involved in the `jmp` instruction. This would otherwise result in an application inconsistent processor state, and might potentially produce an undefined behaviour. I have rather adopted a different approach where the original indirect-branch instructions are substituted at compile-time by PM with so-called

offset jumps (not relying on CPU registers), where the destination address is maintained inside one field of the instruction, and is appropriately set by the on-the-fly correction mechanism. To support the rewrite operation of the appropriate instruction field at run-time, without impacting typical settings associated with memory protection, the offset-jump operation has been moved inside a run-time re-writable ELF section. Also, a jump-label instruction has been inserted in place of the offset jump inside the original (non-writable) sections of the application code, which passes control to the offset jump right after the `brach_corrector` module has re-written the correct destination address (the offset) inside the ad-hoc re-writable section.

## 3.2 Results in the Context of Event-Driven Programming

As mentioned in Section 1, the Event-Driven Programming paradigm [72] models the execution of the program as a flow of events which represent interactions among components of the system, and produce alterations in the actual program's state. In particular, the notion of events is associated with a (logical) timestamp to take care of their logical flow, and the programming model is usually realized by the implementation of *callback* functions which are activated whenever a particular event is scheduled for execution. In the context of speculative execution, like the one provided by optimistic simulation environments [61], the problem of supporting transparency for the incremental execution is non-trivial, since it requires the simulation platform to extract information about which portions of the simulation state are being modified during events' execution. Unlike the proposal in [105], in Section 3.2.1 I present a solution for a completely transparent support for incremental state saving and restore for private portions of the simulation state. This solution is augmented, as described in Section 3.2.2, to account for performance enhancements by relying on the autonomic computing paradimg. Later, in Section 3.2.3, I also face this problem giving a support for shared portion of the simulation state, as well.

### 3.2.1 Supporting Transparency for Private Data Management

In [10], extending the work in [102], I have implemented a memory manager for handling private simulation data explicitly recorded in the heap, in the context of PDES simulations. In particular, the PM tool described in Section 3.1 has been successfully exploited, creating the runtime support to incremental logging. In particular, every memory-write instruction inside the application-level code, namely `mov` instructions with a memory location as the destination, have been modified inserting before each of them a `call` instruction to a provided `update_tracker` module, written in assembly language, which performs the identification of the exact memory address and the size (amount of bytes) involved in the memory update operation. Although this is a typical way for tracking memory update references (e.g. in the context of program debugging techniques [104]), the usage of this approach in optimistic simulation systems poses (more) stringent performance issues. In particular, the monitor should likely perform its job via very few machine instructions, in order not to significantly impact event execution latency.

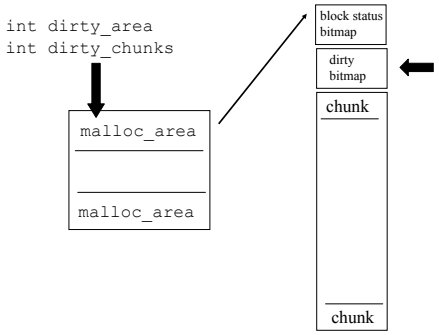
To cope with such a performance target I have explicitly discarded run-time disassembling of the memory reference instruction, which could be too much costly (compared to the event execution latency of non-instrumented software) especially due to the complexity and variable format/length of the Intel instruction set. Instead, I have adopted an orthogonal technique where information related to a particolar `mov` instruction is immediately pushed into the stack by another instruction injected into the executabe during the compile-time instrumentation process. This acts as a sparse cache of disassembling results for memory-write instructions.

In particular, a structure like the one used to correct indirect branches is populated during the instrumentation phase, and it is pushed into the stack at runtime. Additionally, since the opcode of the instruction being statically disassembled, together with its prefixes, establish the real size of the memory area touched by the write operation, this information is also pushed into the stack, and tells `update_tracker` the (compile-time defined) size of the memory area to be dirtied by the current memory-write instruction[3].

So, upon its activation, `update_tracker` checks inside its own stack frame the information needed to compute at runtime the memory address for the write operation and the size of the memory being dirtied. Given that this computation can unpredictably change the value of the EFLAGS register on board of the CPU, this register value is saved by `update_tracker` upon its activation together with general purpose ones, and is put back in place right before returning control to the memory write instruction for which the tracking process has been activated.

[3]The only exception is for `movs` and `stos` instructions, used for moving arbitrary size memory blocks. These instructions keep the information for identifying the destination address and the current size of the memory block being written into predefined registers, namely EDI and ECX, which are directly accessible by `update_tracker`.

Figure 2: Di-DyMeLoR's Memory Map




In the memory model offered by DyMeLoR [102], the subsystem upon which the work in [10] is built, locations associated with automatic variables (allocated inside the stack) do not belong to the object memory map, since they do not survive across different invocations of the event handler. Hence, all those memory-write instructions that can be detected at compile-time to access the stack (e.g. `mov` instructions addressing memory via base pointer or stack pointer displacement) are not actually instrumented by PM, by relying on a special configuration rule. Anyway, in some cases write access into the stack cannot be recognized at compile time. For this reason, after having computed the address for the memory-write operation, `update_tracker` compares it with the current value of the stack pointer. In case the access is an actual stack update, `update_tracker` simply returns. Otherwise, the information about the identified memory address and the size of the area being dirtied is passed to the memory map manager.

A second important aspect presented in [10] is related to the memory management subsystem, which allows to transparently handle the memory allocation requests by the application-level code, and manage the memory map adding the possibility to silently perform some operations—in the case in point, the rollback operations—which enhances the global throughput in specific execution scenarios.

In particular, upon simulation startup, a memory map like the one depiceted in Figure 2 is set up. A `malloc_area` maintains a set of metadata for describing the state of a contiguous memory region which is used to serve memory requests of a given size. In particular, differen `malloc_areas` handle memory chunks of different power-of-two sizes. Two bitmaps, namely the status and the dirty bitmaps, are used to track the current state of memory allocation.

Calls to standard malloc library's APIs are redirected at compile time to DyMeLoR's ones, which upon allocation requests via a `malloc` call selects the best `malloc_area` which allows to serve the memory request (minimizing fragmentation) and in case some chunks are still available, the corresponding bit in the status bitmap is flagged, and the chunk is delivered to the application-level software. If, on the contrary, the status bitmap states that the contiguous region is full, a new `malloc_area` is allocated, along with the bitmaps and the contiguous chunks, and is linked to the previous one.

Via the exploitation of the `dirty_area` and `dirty_chunks` fields inside each `malloc_area`, and of the dirty bitmaps, logging activities performed by Di-DyMeLoR have been differentiated in full and incremental logs. Both types of logs result in packing the information to be logged inside a contiguous buffer allocated via the underlying malloc services. However, they pack different things. A full-log operation implies that the active `malloc_area` entries are packed inside the log buffer together with the in-use chunks in the corresponding memory blocks, while the dirty bitmaps are not logged. On the other hand, an incremental log performs differentiated operations depending on the current value of the control flags/fields. Specifically, for each active `malloc_area` entry we have the following cases:

A: `dirty_area` is set and `dirty_chunks` is zero. In this case the `malloc_area` is packed into the log buffer together with the status bitmap indicating the current allocation of chunks inside a given block. But the dirty bitmap and the currently in-use chunks are not logged.

B: `dirty_area` is set and `dirty_chunks` is greater than zero. In this case the `malloc_area` is packed into the log buffer together with the status bitmap, the dirty bitmap and the chunks that are currently in use,

which have been dirtied.

C: `dirty_area` is not set. In this case, no information associated with the area is logged at all.

Full and incremental logs both involve the re-set of all the data structures tracking dirty data/meta-data. For incremental logs, this occurs independently of the actual case among the aforementioned ones.

Each log is stamped with the current simulation time, and all the logs (full and incremental) are linked together as a chain. When a restore operation needs to be executed at simulation time $T$, the log chain is searched to determine the more recent log with time less than or equal to $T$ (logs with time greater than $T$ are simply discarded since they refer to causally inconsistent memory maps). In case the log found is a full one, then a restore operation is executed by simply unpacking all the logged data and putting them back in place. A different restore algorithm is executed in case the log found is an incremental one. Specifically, the following steps are iterated by backward traversing the chain of logs:

1. A `malloc_area` found inside the log buffer, which has not been restored, is put back in place inside the meta-data table. The associated status bitmap is also copied back from the log buffer (recall that independently of the type of log and of the specific case for incremental logging, a logged `malloc_area` is always associated with the corresponding status bitmap inside the log buffer to guarantee recoverability of chunk allocation/deallocation operations).

2. Each dirty chunk found inside the log and associated with the `malloc_area`, which has not yet been restored in a previous iteration while backward traversing the log, is copied back in its correct position inside the corresponding memory block.

The iterative restore procedure stops when all the active `malloc_area` entries have been restored and all the in-use chunks that have been dirtied are also restored. Although in principles this could entail an indefinite number of iterative backward steps along the log chain, in practice the restore operation can be immediately finalized once we find a full log while backward re-traversing the log chain. In fact, all the in-use chunks that have not yet been restored are immediately available inside the full log for copy-back operations. Actually, to optimize the detection of already restored chunks, which must therefore not be copied-back again from the log, the iterative restore procedure has been based on temporary bitmaps (each associated with an active `malloc_area`) on which a couple of fast bitwise OR-XOR operations are executed each time a dirty bitmap (associated with that same `malloc_area`) is extracted from the incremental log.

In Figure 3.2.1, I present some performance results evaluated using Personal Communication System (PCS), a parameterizable cellular system simulator, explicitly modeling fading and channel interference phenomena, entailing the computation of the minimum transmission power allowing the currently setup call to achieve the threshold-level SIR value, according to GSM technology. In the experimental configuration, the call inter-arrival frequency has been varied, in order to actually show how the memory manager scales when the execution load is increased.

In particular, the proposal has been show to add a very reduced—and constant— overhead (see Figure 3(b)), while at the same time showing a log latency which grows linearly, while the original (non-incremental) latency was showing an exponential trend (see Figure 3(a)). At the same time, the cost for restoring a log is higher in the case of an incremental log (see Figure 3(c)), altought this operation is seldom executed, and the actual cost can be tuned by forcing the interval of the incremental log. At the same time, the memory usage in the incremental version is definitely minimal, even more if compared to the non-incremental one (see Figure 3(d)), giving the possibility to follow through more complex simulations even on machines where the available memory is limited.

### 3.2.2 An Autonomic Approach based on a Dual Coding Mechanism

In [8] an extension to [10] has been presented, in which multiple (differently instrumented) versions of the same executable coexist in the same image. In particular, this work has allowed complex simulation systems to switch between incremental and non-incremental executions of the log/restore operations depending on the actual application-level execution dynamics, thus enhancing the overall throughtput by selecting the most suited execution mode.

Automatic ELF rewriting schemes have been introduced into the aforementioned PM tool, in order to create, starting from the same set of application level modules, two different .text sections within the ELF, one containing a non-instrumented version of the compiled modules, and the other one containing the instrumented counterpart.

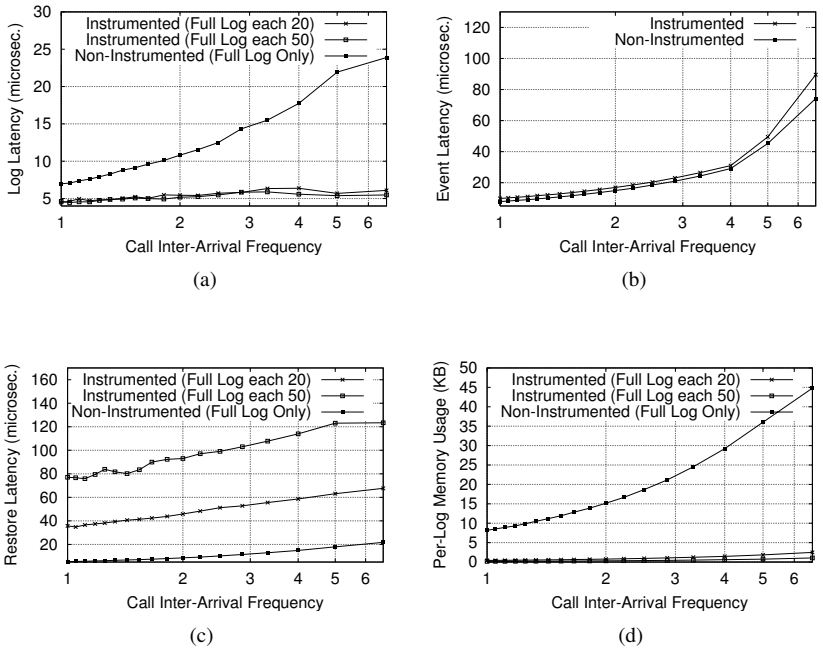



Figure 3: Di-DyMeLoR Performance Results

These two sections are then transparently placed within different virtual memory sections thanks to standard `ld` facilities. However, the corresponding symbol tables are modified by my preprocessing/instrumenting tool in order to expose the application interface requested by the underlying simulation kernel, namely the event handler callback, via differentiated symbols. The .rodata sections corresponding to the two different text sections are modified in order to provide correct adjustment of the displacement information associated with the position of code and data within the virtual memory addressing. Also, the replicated .data/.bss sections associated with the two versions of the application object code have been collapsed on the same virtual addressing range in order to provide a single actual copy of initialized and non-initialized data, accessible by both the generated code versions. A schematization of the whole process supporting such a dual-version code generation is provided in Figure 4, where I explicitly indicate the steps carried out by my extension to PM.

Once the executable is finally built and run, a kernel level switch between the two different log modes simply involves reassigning the event-handler callback pointer to the entry point symbol associated with the corresponding version of the duplicated application executable modules. Adopting this solution, either full or incremental log mode is supported[4] according to an optimized run-time scheme where any overheads are at all avoided while processing simulation events in case no tracking of memory update operations is requested by the currently active log mode.

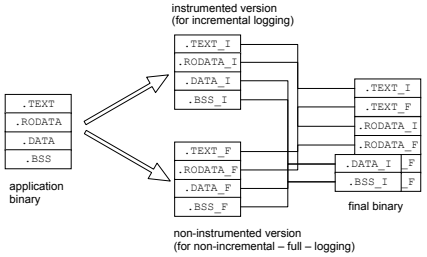
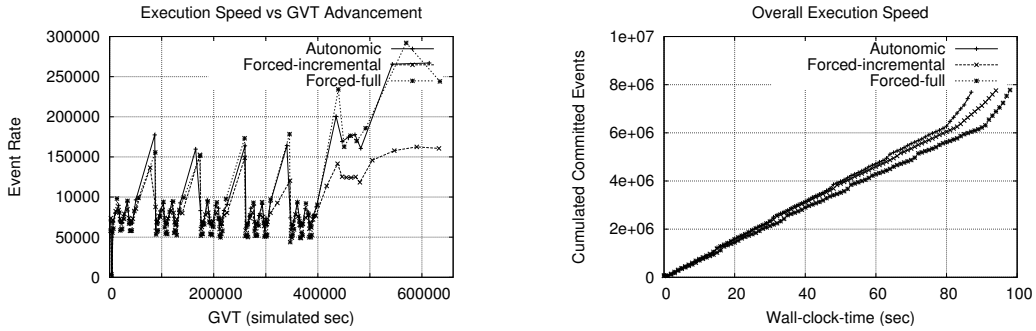



Figure 4: Dual-Code Coexistance Scheme

The above scheme would only entail additional virtual addresses consumption due to the presence of two versions of the executable modules associated with the application layer. However, this should not represent a real problem when considering the tendency of vendors towards 64-bit processors, enabling extremely wide span of virtual memory addressing, and the fact that text sections usually fill a reduced percentage of the available virtual addresses.

I have conducted an experimental assessment of my proposal, by relying on the PCS simulation model, where I have simulated a whole week of operativity of the GSM coverage system along the roman GRA highway, explicitly

[4]The work in [8] actually presents as well a closed-formula method to estimate which are the best incremental/full log interval ($\chi_I$ and $\chi_F$, respectively) according to the current execution dynamics. For the sake of brevity, I refer directly to that work for that part of the discussion.




(a) Event Rate

(b) Throughput

Figure 5: Autonomic System Experimental Results

accounting for dynamic day-time traffic variations, like night and rush hours, as derived by [16].

I report in Figure 5(a) the variation of the amount of committed events per wall-clock-time second (event rate) achieved while simulating specific virtual time periods, represented by the variation of the GVT on the x-axis. Actually, this parameter indicates the speed according to which a given virtual time period is simulated. The higher the event rate, the faster the execution while simulating a given virtual time period. I report three plots referring to (i) the case in which the autonomic layer is active (ii) the case in which the autonomic layer is active, but I always force the incremental log/restore mode, with the corresponding optimized value for $\chi_I$ and (iii) the case in which the layer is active but the non-incremental (full) log/restore mode is forced, with the corresponding optimized value for $\chi_F$. The plots for cases (ii) and (iii) express performance levels that could be achieved via an optimized log/resotre mode (adaptive in the selection of the log interval) based on either the incremental or the non-incremental log mode, but not allowing autonomic switch between the two modes on the basis of run-time dynamics.

By the results, we see that, depending on the simulated period (night-time vs day-time), forced-incremental and forced-full modes alternately exhibit better execution speed. In particular, the forced-full mode is faster while simulating night-time periods, while the forced-incremental mode is faster while simulating day-time periods. This is a reflection of the fact that, during night-times and in the weekend, each GSM cell, and hence each LP, exhibits a reduced state size due to the minimal number of records allocated for ongoing calls. This is not the case for day-time periods, where the state size of the LPs can grow significantly (especially for rush hours), up to the limit of slightly less than 70 KB, and the update pattern of the state upon the occurrence of the events allows the incremental-log mode to outperform the full one, once the correspnding log period get optimized. Anyway, the most important outcome by the event rate plots is that the autonomic configuration always switches to the best performing mode (incremental vs non-incremental) depending on the currently simulated period (e.g. night vs day), and hence depending of the actual dynamics (e.g. in terms of state size, event granularity, memory update pattern and so on).

The final effect on performance by the above optimized behavior is expressed by the plots in Figure 5(b), where we draw the cumulated amount of committed events vs the wall-clock-time for the simulation run. These curves express the ability of each log/restore configuration to commit events (and hence to carry out useful simulation work) while wall-clock-time goes ahead, hence we have a representation of how fast the simulation model is executed vs wall-clock-time. By the results, the ability of the autonomic configuration to always switch to the best suited mode is reflected in the fact that its cumulated event rate curve always exhibits the best pendency vs wall-clock-time. In other words, it allows the model execution to be carried out in a significantly faster manner, compared to what done by the other two schemes. In particular, the wall-clock-time by the autonomic scheme for reaching the required amount of events to be committed for the whole simulation is reduced of about 13% compared to the forced-full mode, and of about 9% compared to the forced-incremental mode. Given that these modes run according to an optimized configuration, thanks to dynamic (re-)selection of well suited log intervals, this is a significant result.

### 3.2.3 Supporting Transparency for Shared Data Management

In [3] I have presented a solution to the problem of handling shared state in optimistic simulation. In particular, according to the seminal paper in [61], the whole simulation state $S$ is partitioned amongst the various $n$ LPs which together set up the simulation, in a way such that (i) $S = \bigcup_{i=1}^{n} S_i$ and that (ii) $\forall i, j \; S_i \cap S_j = \emptyset$. This means that in order to synchronize or communicate, different LPs must rely on explicit events (i.e., message passing). The aforementioned work in [3] specifically addresses this problem by relaxing constraint (ii) and allowing global variables to be consistently accessed during the optimistic simulation's execution.

This solution clearly sets forth the path of transparency, in the sense that it allows programmers to rely on conventional programming facilities (i.e., assignments to global variables), re-maps these operations to different ones better matching multi/many-core architectures, and additionally tackles performance by explicitly relying on a non-blocking implementation.

In order to provide complete transparency to the application-level programmer, accesses in read/write mode to global variables must be explicitly intercepted. To this end, I have again relied on the PM tool, modifying the actual instructions executed by software executables, without altering their actual semantics. By relying on PM, at compile time the application-level instruction code (i.e., the assembly bytestream) is modified in order to replace operations loading data to and from memory with actual function calls which are the entry points of my Shared State Memory Subsystem (SSMS). These entry points are associated with the following APIs provided by SSMS: `write_global_variable(void *orig_addr, time_type lvt, ...)` and `void *read_global_variable(void *orig_addr, time_type my_lvt)`. They allow accessing the versions within the version lists for a given variable at a certain Logical-Virtual-Time (LVT).

I have identified two main groups of instructions/code blocks which have to be handled within the application-level assembly code. First, in IA-32 simple load and store operations are identified by `mov` instructions. Whenever IT's parser identifies a `mov` instruction, it is analyzed in order to determine whether it is targeting memory as a source or destination operand, and a call to `write_global_variable` or `read_global_variable` is replaced accordingly. When the `mov` instruction involves a load operation from memory, an additional postamble to the function call is placed, in order to have the actual value returned by `read_global_variable` placed into the correct CPU register where the application-level software is expecting the value to be found.

Second, the IA-32 instruction set provides more complex instructions which allow an executable to efficiently modify memory areas in-place. As a relevant example, I propose instructions like `ADD m32, r32` or `INC m32`. In this case, IT replaces the instructions with a block of instructions, entailing a couple of calls to the SSMS's read and write APIs, and re-implementing the same logic with several CPU instructions. This implementation of course adds some overhead, nevertheless it allows to integrate my SSMS completely transparently wrt the application-level programmer.

High-level programming languages allow to access memory objects in a non-direct way, namely through the use of pointers. Since IT works at compile time, it is not possible to statically determine whether a pointer will target a global variable or not. To cope with this issue, I use IT to instrument any `mov` instruction which can handle pointers through a call to a `monitor` function which fastly determines if a pointer targets a global variable. In particular, at compile time, via the usage of a custom ld-based linker script I insert symbols called `_bss_start`, `_bss_end`, `_data_start`, `_data_end`, within the application-level ELF executable, which mark off the area containing global variables. Upon a call to the `monitor` routine, a fast check on these boundaries is performed. If a pointer falls within this area, the operation is redirected to SSMS, on the other hand the original `mov` instruction is executed.

As a last note, Intel's instruction set provides *string instructions* which allow to perform operations on memory buffers instead of single memory locations. In particular, `movs` and `stos` instructions allow the program to copy or modify large buffers at once. In order to cope with the presence of these complex instructions, SSMS provides two additional APIs, namely `copy_buffer()` and `set_buffer()` which simulate the execution of these operations on version lists if they are found to target global variables (e.g., global arrays). Otherwise, they just execute the original `movs` or `stos` operations. Therefore, at compile time, IT replaces every string operation involving memory update with a function call to these APIs, accordingly.

The last operation I perform at compile time is the inspection of the application-level ELF object file in order to extract information concerning global variables. In particular, by exploring the application object I extract from the symbol table `.symtab` all the `STT_OBJECT` / `STT_COMMON` symbols and store their name, address and size in a text file which will be later used at startup time for setting up the version lists. In this way, by exploiting the $\langle name, address, size \rangle$ tuple, I am able to transparently identify any access to global variables which will be

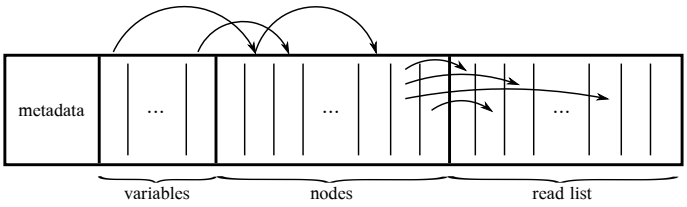



Figure 6: Preallocated Shared Memory Map

likely used by the application-level code during the execution of the simulation model, allowing the programmer to rely on the complete set of constructs provided by ANSI-C. I note that, although there will be more instances of the simulation kernel running the application-level code, a global variables' address is a common information shared among the instances, as long as its virtual address will be the same and is cabled into the executable.

SSMS explicitly targets shared-memory/multi-core machines. In order to significantly enhance performance, I have decided to avoid requesting to the underlying operating system shared memory segments on-demand, whenever SSMS needs to install some data structure. On the other hand, at simulation startup the master kernel installs a large shared memory segment, and broadcasts to other kernel instances its id. The shared segment is partitioned according to the definition of the following structure:

```
typedef struct _globval_shmem {
  int num_vars;
  globvar_info  variables[MAX_GLOBVARS];
  volatile int  first_node_free;
  globvar_node  versions[MAX_VERSIONS];
  time_type read_list[];
} globvar_shmem;
```

In particular, the shared memory segment is divided into several fixed-sized portions. One portion, namely `variables`, is an array which is used to manage global variables. Upon initialization of SSMS, the configuration text file is loaded and parsed. The field `num_vars` is used to keep track of how many variables are actually handled, and for each of them an entry in the `variables` array is populated. To allow a fast retrieval of the global variables, I use a fast hash function to determine which entry in the `variables` array will store the information associated with a specific variable. In particular, the position in the array is determined with a fast bitwise operation — namely, `address & (~(-MAX_GLOBVARS))` — since `MAX_GLOBVARS` is set to be a power of two. In case collisions are found, separate chaining is used as a means for finding a free place. Each entry in the `variables` array is structured as:

```
typedef struct _globvar_info {
  void *orig_addr;
  unsigned short int size;
  long long head;
  long long tail;
} globvar_info;
```

`orig_address` stores the global variable's original address, which is used as hash table's key; `size` describes which is the size (in bytes) of the global variable.

Since I am preallocating shared memory, version lists must be implemented using nodes scattered around the preallocated segment. In particular, `versions` is an array of fixed-sized nodes which can be used for any list, and `head` and `tail` are indices within this array, which is composed of entries structured as follows:

```
typedef struct _globvar_node {
  volatile int alloc;
  time_type lvt;
  unsigned char value[MAX_BUFF];
  spinlock_t read_list_spinlock;
  long long next;
} globvar_node;
```

where `lvt` is the logical time associated with the version, `value` is the global variable's value, and `next` is used to identify which is the following node in the list. A node can therefore be seen as a snapshot of the state of a single global variable at a certain LVT. In Figure 6 I provide a complete picture of the preallocated memory map.

Node versions' entries can belong to any list, and given that lists are accessed without the use of locks, a special allocation function must be used, ensuring that no two simulation kernel instances running concurrently are given the same entry for handling two different versions.

**Algorithm 1** Shared Memory Allocation

```
 1: procedure ALLOCATE
 2:     m ← generate_mark()
 3:     slot ← first_node_free
 4:     while true do
 5:         alloc ← vers[slot].alloc;
 6:         if alloc ∨ ¬ CAS(vers[slot].alloc, alloc, m) then
 7:             slot ← next slot in circular policy
 8:         else
 9:             break
10:         end if
11:     end while
12:     atomically update first_node_free
13:     return slot;
14: end procedure
```

The ALLOCATE pseudocode is given in Algorithm 1. In order to allow concurrent accesses, it relies on `CAS`[5], which allows to update involved data only if no other process has updated the same data in the meanwhile. The `globvar_shmem` data structure holds in `first_node_free` the value of the first element of the `versions` array to start trying to allocate from. Its manipulation is based on the classical algorithm used by the LINUX kernel for managing the bitmap of file descriptors associated with a process. Specifically, it is always atomically increased upon allocation, and gets atomically decreased in case an entry is released having index less than the first chunk currently available within that block. Starting from that slot, a kernel instance tries to allocate a node by storing via a `CAS` operation a non-zero value within the `alloc` field of `globvar_node`, which tells whether a node is currently in use. In case the `CAS` fails, the next node in the array is selected and the procedure is repeated, until it eventually succeeds[6]. The companion function RELEASE is much simpler, as it only entails resetting the `alloc` and updating `first_node_free`, via an `atomic_set` call.

In order to cope with the ABA problem, I have explicitly decided to consider a node allocated if the `alloc` field is non-zero. In particular, I store into it a unique value every time a node is allocated, so that two allocations can be identified as different. The macro `generate_mark` produces an integer value which is in turn composed of two short integers, one holding the unique id of a kernel instance and the other holding the value of a per-kernel counter which is incremented every time the macro is invoked[7].

Once a node is allocated, it gets organized into a non-blocking linked list, which is implemented according to a modified version of the one proposed in [48]. Concurrent insertions are handled via the use of a single `CAS` operation, which is used to introduce the newly allocated node into the list by acting on the `next` field of the predecessor node. As for deletion, two `CAS` are used, one to mark the `next` field of the deleted node as *logically* deleted, and another to *physically* delete the node. I have slightly modified the algorithm in order to take into account my specific needs. In particular, the FIND-NODE procedure has been augmented in order to return the `alloc` field, to explicitly cope with the ABA problem, and the INSERT procedure does not fail if a node with the same key (i.e. LVT) already exists. Specifically, the new node is simply linked after the originally existing one. In addition, I note that LPs are more likely to access versions associated with higher LVTs, since well partitioned/balanced optimistic simulations usually proceed relatively evenly. Therefore, I sort the versions in the lists in descending order, to avoid a complete scan of the list every time we want to find a node in it.

To avoid the ABA problem in linked lists, pointers (i.e. indices) to nodes are composed (every time they are updated) by a unique mark generated via the aforementioned macro `generate_mark` and the real index, allowing to capture the situation where two nodes are still adjacent but one was deallocated and then reallocated during the execution of the non-blocking algorithm by different kernel instances.

The operations performed on the versions lists are depicted in Figure 7(a).

The APIs offered by SSMS provide two main functions to access global variables, namely `read_global_variable` and `write_global_variable`, which I will refer to as READ and WRITE from now on.

READ operation's pseudocode is provided in Algorithm 2. For efficiency reasons, before letting an LP execute a simulation event, SSMS sets up an $AccessSet$, i.e., a mapping between version nodes and variables. Whenever a variable is accessed for the first time, FIND-NODE ([8]) determines which is the most suitable version for the given LVT, and a couple $\langle slot, version \rangle$ is placed into $AccessSet$ in order to speedup the retrieval of the version,

[5] In particular, I rely on the IA-32's `cmpxchg`. I often mention atomic operations, which are implemented directly in assembly using native atomic instructions.

[6] To check if the space is up, a counter of available free nodes is kept as well in shared memory, which is managed via an `atomic_decrement` operation.

[7] `generate_mark` can of course return two equal values when the counter overflows, but this situation can happen after a significant simulation time, so I consider it to be statistically non-significant for the ABA problem.

[8] I remind that FIND-NODE is a modified version of the one presented in [48]. For a detailed description of the procedure, I throw back to that work. In addition, I note that a version node is always available, even before any WRITE operation, since at startup the initial value of the global variable is placed into the version list.

**Algorithm 2** Global Variable Read

```
1:  procedure READ(addr, lvt)
2:      slot ← hash table's entry associated with addr
3:      hasRead ← false
4:      if slot ∈ AccessSet then
5:          version ← AccessSet[slot]
6:      else
7:          while ¬hasRead do
8:              ⟨version, alloc⟩ ← FIND-NODE(slot, lvt)
9:              AccessSet[slot] ← version
10:             spin_lock(read_list_lock)
11:             if alloc has been changed then
12:                 spin_unlock(read_list_lock)
13:                 continue
14:             end if
15:             add ⟨lp, lvt⟩ into ReadList
16:             spin_unlock(read_list_lock)
17:             hasRead ← true
18:         end while
19:     end if
20:     return vers[version].value;
21: end procedure
```

**Algorithm 3** Global Variable Write

```
1:  procedure WRITE(addr, lvt, val)
2:      slot ← hash table's entry associated with addr
3:      if slot ∈ AccessSet then
4:          version ← AccessSet[slot]
5:          vers[version].value ← val
6:      else
7:          version ← INSERT-VERSION(slot, lvt, val)
8:          AccessSet[slot] ← version
9:      end if
10:     for all ⟨lp, lvt'⟩ ∈ ReadList s.t. lvt' ≥ lvt do
11:         send antimessage to lp
12:     end for
13: end procedure
```

avoiding the scan of the list upon subsequent accesses.

As for the WRITE operation, the pseudocode of which is presented in Algorithm 3, its behavior is twofold depending on whether it is invoked for the first time since the beginning of the current event's execution. In particular, upon the first access on a variable, the $AccessSet$ for that particular event is populated. Otherwise, a call to INSERT-VERSION is performed whichcreates a new version. The second part of the WRITE operation entails checking the $ReadList$ for ensuring consistency.

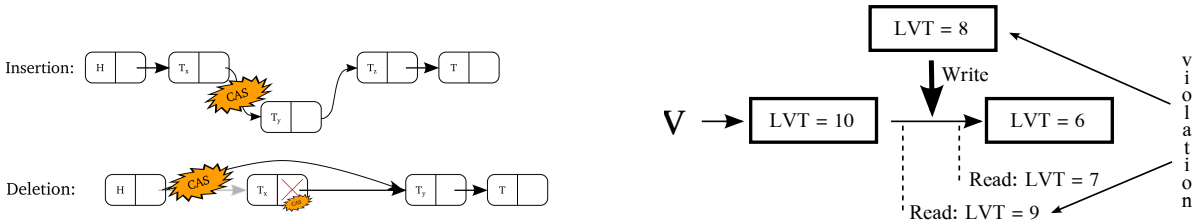
In order to strengthen the optimism of my implementation, I allow interleaved reads and writes on a version list, and I explicitly avoid a version $k$ installed at LVT $t_k$ to invalidate every version $j$ such that $t_k < t_j$. In fact, I note that consistency is violated only if, at LVT $t_x$ an LP reads the version associated with LVT $t_y$ such that $t_y \leq t_x$, and at a certain point during the execution a new version node associated with LVT $t_z$ such that $t_y \leq t_z < t_x$ is installed.

This means that every process which reads a certain version node must leave a mark of that operation, i.e., visible reads [26] are enforced. In fact, as shown in Figure 7(b), I are interested in undoing only the events which read a version older than the new one which has just been inserted.

To this end, I augment the classical notion of rollback as presented by the Time Warp synchronization protocol, by sending a special anti-message to all the LPs which have read a so-defined causally inconsistent version after any write operation. This is reflected into Algorithms 2 and 3. In fact, in the READ operation, before returning the variable's value, the tuple $\langle lp, lvt \rangle$ is inserted into the $ReadList$ for that particular version. This operation is included within a specially designed critical section to ensure consistency. In fact, a spinlock for that particular $ReadList$ is taken, ensuring that no other process will start the rollback operation while the $ReadList$ is being updated. Otherwise, this scenario would produce a non-trackable read operation. In addition, after the spinlock has been taken, a check on the variation of the `alloc` field for that particular version is performed, so to avoid the ABA problem due to a critical race between the deallocation/allocation procedure and the $ReadList$ update. At the same time, at the end of the WRITE operation, the $ReadList$ of the left node is checked in order to find all the LPs which read the previous node's value, while they were requesting a version at an LVT such that they should have read the one in the version which was just installed. Although the list is linked in only one direction, given the implementation of FIND-NODE, locating the previous node is immediate.

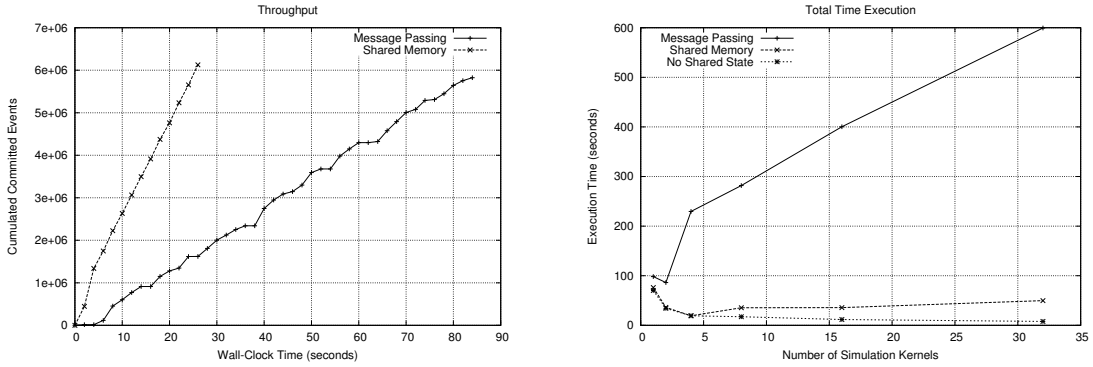
I note that another step must be undertaken in order to ensure correctness. In particular, whenever a special antimessage is received because of an inconsistent read, any version node installed due to that particular event must be removed. To this end, I augmented the concept of *message queue* and modified the WRITE function so that whenever a node is installed during the execution of an event, the message queue keeps track of this operation via a pointer to the node created during the event's execution. In case a rollback operation entails the undoing of that event, the node is removed from the version list, and the $ReadList$ is scanned for sending antimessages to every LP which read that particular node.

In Time Warp, the notion of *fossil collection* is defined, i.e., the process of recovering memory by deleting simulation state snapshots which are no longer needed. In particular, at a periodic rate, the Global Virtual Time (GVT) is computed as the minimum timestamp of not yet processed events or in-transit messages/antimessages in




(a) List Operations (b) Causality Violation

Figure 7: Multiversioned Variables




(a) (b)

Figure 8: SSMS Performance

the whole simulation system. Since during the execution of an event an LP can schedule a new event at an LVT which is equal to, or greater than, the one associated with the event being executed, there cannot be a rollback operation involving a simulation state snapshot associated with a timestamp less than the GVT. Therefore, any snapshot belonging to a logical time window before the GVT can be discarded.

In my proposal, I extend the notion of *fossil collection* by defining the *version list pruning*. In particular, upon GVT computation, the version lists associated with global variables are scanned in order to find which is the first node $i$ stamped with $t_i \leq GVT$ and that node is selected as the barrier node. Any node marked with a timestamp $t_k < t_i$ is marked as free and removed from the list. For implementations where there is no actual event processing during GVT computation, the version list pruning is thread safe, and can therefore be executed efficiently, with no need to synchronize the access. In particular, the various lists can be divided evenly across the various kernel instances, and each kernel performs the memory recover executing in isolation. This choice provides a more efficient execution and still ensures correctness.

To evaluate the efficiency of my proposal, I have extended the PCS simulation model having a set of global variables handling global statistics. In particular, upon each event's execution the total number of calls, the total number of handoffs, and the global cumulated power is updated in the shared state. In addition, I have re-implemented a different version of the model in order to have a centralized LP keeping in its disjoint simulation state the global attributes. Every LP willing to update a shared attribute issues a message request to the centralized LP, which in turn sends back the current value. Any update on the current value is then sent as another message to the centralized LP.

In Figure 8(a) I present the throughput associated with my proposed test-bed model run on top of 32 simulation kernel instances, each one running on a private CPU-core of my test machine. By the results, I can see that the execution of the simulation model relying on my SSMS provides a speedup in the order of 70%. In addition, I note that there is a tangible difference between the two curves' trends. In fact, the throughput associated with the SSMS execution has a constant growth, which suggests a constant event commitment rate. On the other hand, the centralized-LP implementation's slope shows fluctuations, which are related to the large amount of events associated with variables' reads/updates which must be processed. Therefore, the number of committed events per GVT interval is not constant, due to the fact that the amount of workload processed by differentiated LPs is

totally different and that the LVT of the LP keeping the shared state diverges from the other LPs' one (this can entail a higher rollback probability), a scenario which is not present at all when relying on the multiversion lists in the shared memory version case.

At the same time, Figure 8(b) shows the total execution time of the simulation wrt the number of parallel simulation kernel instances on which the model is run. In addition to the set of experiments described before, I present also the curve associated with another implementation of the benchmark, where the shared attributes are kept in the disjoint LPs' simulation states and are reduced at the end of the simulation. By the results, I can see that both the SSMS and the centralized-LP implementation suffer from some form of thrashing. In fact, the centralized-LP version provides a speed-down in the order of 100% when the model is parallelized on top of 4 parallel kernel instances, while SSMS shows the same behaviour starting from 8 parallel kernel instances. The version with no shared state shows a trend which is the one expected by a parallel simulator.

I note that in this configuration, the SSMS's speedup wrt the centralized-LP is very large. Of course, the overhead in the centralized-LP case could be leveraged by having different LPs handle different variables, but this solution would not scale well wrt the size of the shared state in the simulation model.

Finally, I note that the simulation model used to assess the validity of my proposal is a worst case for my architecture, since at every event's execution some updates on the global variables are performed, producing a large contention on the linked lists. A simulation model which relies on shared-state for synchronization rather than for global statistics would benefit much more from the proposed architecture.

## 3.3 Achieved Results in the Context of Software Transactional Memories

In an ongoing work [15], I am addressing the problem of performance in the context of high-contention executions of transactional programs. In particular, given the fine synchronization granularity provided by transactions, it is important to study the viability of developing a scheme targeted at undoing the smallest amount of work done as possibile.

In particular, it is evident that whenever a transaction is aborted, it is because some object accessed in read is detected not be consistent (since, e.g., some other transaction has modified it as well, and the read value is no longer valid). To this end, in [15] I have relied on the *snapshot extension* [37] protocol, that is, whenever a conflict is detected (i.e., some object accessed by a transaction $T$ is associated with a logical clock value higher than the transaction's) instead of aborting the whole transaction, I try to read again the new value of the object and check if the previously read ones are still valid. If this is not the case, then the partial rollback rollback algorithm is triggered, and the computation is restarted from the first accessed value in $T$ which is no longer valid.

I emphasize that this approach involves a relation change between the transaction and the snapshot. In fact, in the traditional abort scheme a transaction was allowed to look at a snapshot related to a fixed logical clock's time. In my proposal, on the other hand, the snapshot which is seen by a transaction is dynamically re-evaluated in order to make it aware of possible changes happening during its execution. In particular, the partial rollback approach ensures that, given the fine-grained locking scheme used, we discard only the minimum portion of the work which is needed to restore the execution into a consistent state, thus mixing objects' values which are related to different logical values of the global clock.

To successfully support this approach, two basic aspects must be faced: (i) the CPU state must be restored to the one which was found at that particular point in the transaction (a problem which is addressed by traditional STM systems, which restart the execution of a transaction from its beginning) using the standard `sigsetjmp` library function, and (ii) the stack state must be restored as well. This is particularly important, since normally STM systems address only the consistency of shared objects, while they completely discard the state of per-thread objects. If, on the contrary, a thread's execution must be restored, e.g., in the middle of a function call, we must guarantee that non-consistent updates of local portions of per-thread data (i.e., automatic variables) are restored as well.

To this end, I have relied again on PM, using a memory-update tracking mechanisms similar to the one described in Section 2, yet tuning the assembly monitoring routine to just detect stack changes (i.e., changes in functions' automatic variables). An ad-hoc module for logging versions of automatic variables, based on fast hash tables, has been developed, optimizing in particular the "unbounded" nature of the number of automatic variables which can be found in a program's function. Whenever a partial rollback is triggered, the log chain is backward traversed, restoring the older automatic variables' values, until the target point in the execution which we want to restart from is reached.

In Figure 9 I present a preliminar result of the aforementioned architecture for supporting partial rollback, where a synthetic benchmark has been used in order to assess the actual overhead of our monitor/log/restore architecture. In particular, this benchmark includes within a transaction updates to a local (per-function) variable-sized array, which is filled with data taken from a set of shared objects. With a certain random probability, shared objects are updated as well during this procedure, forcing some transactions to abort. This transaction is included within a cycle, which forces threads to repeatedly execute this process. To study the actual overhead, I have explicitly varied the size of this local array, relying on 6 concurrent threads. Two curves are shown, one which presents the actual execution time when running with the traditional rollback scheme, and one relying on the partial rollback scheme. Both curves are averaged over 10 different runs. By this preliminary results, we can see that the actual overhead induced by our monitoring mechanism is minimal, and mostly constant with respect to the actual size of the local data being used.

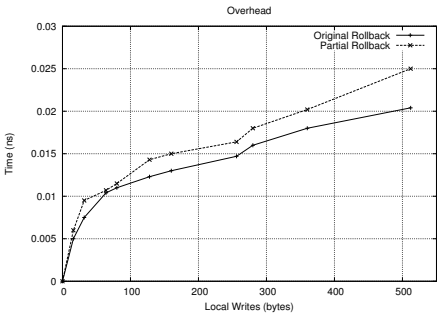


Figure 9: Partial Rollback Overhead

### 3.4 Dissemination of the Results

All my research results have debouched firstly into prototypes, and have later become part of an open-source fully-featured optimistic simulation engine, namely the ROme OpTimistic Simulator (ROOT-Sim) [86] [6], which is publicly available online and which can be downloaded and straightly used by any researcher to follow through his activities.

## 4 Open Issues and Future Work

The main focus of my future work will be mainly on *interaction transparency* and *deploy transparency*, besides on transparency with respect to the *programming model*. In particular, I will concentrate on the following aspects:

- *Interaction transparency*: One of the most important features which programmers want their software to be able to support, is to produce output, i.e. to make the parallel program interact with other worlds, which are not necessarily parallel as well. Output is recognized as a non-trivial problem in the context of parallel programming, especially when dealing with optimistic/speculative executions. This is related to the operations which should be performed when the user is asking to produce some output starting from not-yet-committed data. In particular, since outputting is generally speaking a non-rollbackable operation (in fact, if the output is produced on the screen, on some network device, or on some external device in general, this operation cannot be undone), great care must be used to decide whether the user's request must be fullfilled or not. Additionally, this decision must be done efficiently, since all the time spent in this internal management procedure is time substracted from the actual execution of the program, which is the main goal of a speculative execution.

  In my future work, I want to study the viability and efficiency of different possibilities (i.e. block-until-commit and wait-until-commit) as a support to a consistent and efficient parallel output module, both in the multi/many-core and in the distributed domains.

  As for the input problem, the main goal is to study in the context of speculative/optimistic executions which are the implications of giving the user the possibility to interact with the ongoing flow of the execution. In particular, if we look at the context of optimistic simulation, consistency is guaranteed by using the speculative retry-until-commit approach. Yet, in order to efficiently use the available computing resources (in particular memory) whenever an event is considered committed, every information associated with it (e.g., state logs) are discarded. If the user, during the flow of the simulation, produces an interaction associated with a committed portion of the simulation, the engine supporting it might not be able to handle the user's request. My main goal is to study the most effective tradeoff between the user's possibility to interact at any time of the execution of the simulation, and the actual efficient resource management which such enviroments must guarantee.

- *Deploy transparency*: On the path of seeking transparency, facing this problem entails the desing/development of tools and techniques which address the selection of the most suited amount of parallel computing resources, in order to minimize thrashing due to synchronization operations. In fact, if we support the end user with a transparent environment which allows to rely on a set of facilities unlocking concurrency, due to the fact that he is not expected to be an expert of the field, at the same time selecting the best amount of computing resources needed to efficiently follow through the execution might be a difficult task as well. This is particularly important if we set ourselves on the modern trend of, e.g., Cloud Computing, where requesting more resources than needed might not only entail a thrashing in the execution, but a money/resources waste as well.

- *Programming Model Transparency*: Concerning this point, the core aspect focuses on a portion of end users which is not necessarily completely unaware of parallelism. In fact, if dealing with supporting the parallelization of, e.g., an event-driven program the end user specifies in its source code that he is willing to explicitly exploit some parallelism (e.g., by explicitly relying on multithreading by calling functions from the pthread library), the question would be how to map this user request on top of the speculative framework which support the parallel execution of its software.

  In my future work, I want to study the viability of several approaches targeted at mixing together the notion of parallelism provided by the end users, and the parallelism induced by the executiong framework which I have been working so far.

# 5 Publications

## 5.1 Conference Articles


[1] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. A load sharing architecture for optimistic simulations on multi-core machines. In *Proceedings of the 19th International Conference on High Performance Computing*, HiPC. IEEE Computer Society, December 2012. To Appear.

[2] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Assessing load sharing within optimistic simulation platforms (invited paper). In *Proceedings of the 2012 Winter Simulation Conference*, WSC. Society for Computer Simulation, December 2012. To Appear.

[3] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. Transparent and efficient shared-state management for optimistic simulations on multi-core machines. In *Proceedings 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS, pages 134–141. IEEE Computer Society, August 2012.

[4] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Towards symmetric multi-threaded optimistic simulation kernels. In *Proceedings of the 26th International Workshop on Principles of Advanced and Distributed Simulation*, PADS, pages 211–220. IEEE Computer Society, August 2012.

[5] Roberto Vitali, Alessandro Pellegrini, and Gionata Cerasuolo. Cache-aware memory manager for optimistic simulations. In *Proceedings of the 5th International ICST Conference of Simulation Tools and Techniques*, SIMUTools, March 2012. Winner of the Best Paper Award.

[6] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. The ROme OpTimistic Simulator: Core internals and programming model. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools. ICST, 2011.

[7] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. An evolutionary algorithm to optimize log/restore operations within optimistic simulation platforms. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools. SIGSIM, 2011.

[8] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Autonomic log/restore for advanced optimistic simulation systems. In *Proceedings of the Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS, pages 319–327. IEEE Computer Society, 2010.

[9] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Benchmarking memory management capabilities within root-sim. In *Proceedings of the 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*. IEEE Computer Society, 2009.

[10] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, PADS, pages 45–53. IEEE Computer Society, 2009. Candidate for (but not winner of) the Best Paper Award.

## 5.2 Talks Given

[11] Alessandro Pellegrini. A symmetric multi-threaded architecture for load-sharing in multi-core optimistic simulations, July 2012.

# 6 Submitted / In-Preparation Papers

## 6.1 Journal Articles

[12] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. A symmetric multi-threaded architecture for load-sharing in multi-core optimistic simulations. *ACM Performance Evaluation Review*. Fast Track invitation as InfQ 2012 Selected Paper. In Preparation.

[13] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Autonomic state management for optimistic simulation platforms. *IEEE Transactions on Parallel and Distributed Systems*. In Preparation.

## 6.2 Conference Articles

[14] Alessandro Pellegrini and Giuseppe Piro. Multi-threaded simulation of 4G cellular systems within the LTE-Sim framework. In *Proceedings of the 8th IEEE International Workshop on the Performance Analysis and Enhancement of Wireless Networks*, PAEWN. IEEE Computer Society, March 2013. Under Review.

[15] Alice Porfirio, Alessandro Pellegrini, Pierangelo Di Sanzo, and Francesco Quaglia. Efficient partial rollback in software transactional memories. In *Preparation for Submission to the 22nd Conference on Compiler Construction*, 2013.

# 7 References


[16] `http://traffico.octotelematics.it/`.

[17] GDB: The GNU Project Debugger. `http://www.gnu.org/software/gdb/`.

[18] *System V Application Binary Interface AMD64 Architecture Processor Supplement*, December 2007.

[19] Umut A. Acar. Self-adjusting computation: (an overview). In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM, pages 1–6. ACM, 2009.

[20] Umut A. Acar, Guy Blelloch, Ruy Ley-Wild, Kanat Tangwongsan, and Duru Turkoglu. Traceable data types for self-adjusting computation. *SIGPLAN Not.*, 45(6):483–496, June 2010.

[21] Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. *SIGPLAN Not.*, 41(6):96–107, June 2006.

[22] S. Akhter and J. Roberts. *Multi-Core Programming: Increasing Performance Through Software Multi-threading*. Books by engineers for engineers. Intel Press, 2006.

[23] James H. Anderson and Yong-Jik Kim. A new fast-path mechanism for mutual exclusion. *Distrib. Comput.*, 14(1):17–29, January 2001.

[24] Azer Bestavros and Spyridon Braoudakis. Value-cognizant speculative concurrency control. In *Proc. of VLDB*, pages 122–133, 1995.

[25] M.J. Bridges, N. Vachharajani, Yun Zhang, T. Jablin, and D.I. August. Revisiting the sequential programming model for multi-core. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 69–84, dec 2007.

[26] James Burns and Nancy A. Lynch. Mutual exclusion using invisible reads and writes. In *Proceedings of the 18th Annual Allerton Conference on Communication, Control, and Computing*, pages 833–842, 1980.

[27] Richard P. Case and Andris Padegs. Architecture of the IBM system/370. *Communications of the ACM*, 21:73–96, January 1978.

[28] J.D. Collins, Hong Wang, D.M. Tullsen, C. Hughes, Yong-Fong Lee, D. Lavery, and J.P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 14 –125, 2001.

[29] Intel Compiler. https://computing.llnl.gov/?set=code&page=intel_profiler.

[30] Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. Reactive imperative programming with dataflow constraints. In *Proceedings of the Object-Oriented Programming, Systems, Languages & Applications Conference*, OOPSLA, pages 407–426. ACM, October 2011.

[31] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, pages 194–208, 2006.

[32] DynamoRIO. http://www.dynamorio.org/.

[33] DynInst. http://www.dyninst.org/.

[34] William B. Easton. Process synchronization without long-term interlock. *SIGOPS Operating Systems Review*, 6(1/2):95–100, June 1972.

[35] EEL. http://pages.cs.wisc.edu/∼larus/eel.html.

[36] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC, pages 47–. IEEE Computer Society, 2004.

[37] Pascal Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-based software transactional memory. *Parallel and Distributed Systems, IEEE Transactions on*, 21(12):1793–1807, December 2010.

[38] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[39] Josef Fleischmann and Philip A. Wilsey. Comparative analysis of periodic state saving techniques in time warp simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 50–58. IEEE Computer Society, June 1995.

[40] Steve Franks, Fabian Gomes, Brian Unger, and John Cleary. State saving for interactive optimistic simulation. In *Proceedings of the 11th workshop on Parallel and Distributed Simulation*, pages 72–79. IEEE Computer Society, 1997.

[41] Keir Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, September 2003. Also available as Technical Report UCAM-CL-TR-579.

[42] Frysk. http://sourceware.org/frysk/.

[43] Richard M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[44] Thorsten Grotker. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[45] Rachid Guerraoui, M.K. Ka, and N. Lynch. *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2011.

[46] Tsuyoshi Hamada and Keigo Nitadori. 190 tflops astrophysical n-body simulation on a cluster of gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, pages 1–9. IEEE Computer Society, 2010.

[47] Matthew A. Hammer, Umut A. Acar, and Yan Chen. CEAL: a C-based language for self-adjusting computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2009.

[48] Timothy Harris. A pragmatic implementation of non-blocking linked-lists. In Jennifer Welch, editor, *Distributed Computing*, volume 2180, pages 300–314. Springer Berlin / Heidelberg, 2001.

[49] Timothy Harris and Keir Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38(11):388–402, October 2003.

[50] Shenin Hassan, Dhiya Al-Jumeily, and Abir Jaafar Hussain. Autonomic computing paradigm to support system's development. In *Proceedings of the 2nd International Conference on Developments in eSystems Engineering*, DESE, pages 273–278. IEEE Computer Society, December 2009.

[51] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[52] Maurice P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13:124–149, January 1991.

[53] Maurice P. Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, pages 92–101. ACM, 2003.

[54] Maurice P. Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA, pages 289–300. ACM, 1993.

[55] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[56] Paul Horn. Autonomic computing: IBMs perspective on the state of information technology. 15:1–39, 2001.

[57] Intel Corporation. *IA-32 Intel(R) Architecture Software Developer's Manual Volume 1: Basic Architecture*.

[58] Intel Corporation. *IA-32 Intel(R) Architecture Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M*.

[59] Intel Corporation. *IA-32 Intel(R) Architecture Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z*.

[60] Intel Corporation. Enhanced ıntel SpeedStep technology for the ıntel Pentium M processor. `http://download.intel.com/design/network/papers/30117401.pdf`, March 2004. White Paper.

[61] David R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and System*, 7(3):404–425, July 1985.

[62] Eric H. Jensen, Gary W. Hagensen, and Jeffrey M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. (Technical Report UCRL–Technical Report97663), November 1987.

[63] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41 – 50, January 2003.

[64] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP, pages 223–234. ACM, 2011.

[65] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. *SIGPLAN Not.*, 47(8):141–150, February 2012.

[66] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866 –880, September 1999.

[67] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[68] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, January 1987.

[69] M.A. Laurenzano, M.M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 175–183, March 2010.

[70] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. Signal–a data flow-oriented language for signal processing. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 34(2):362–374, apr 1986.

[71] Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for java. *ACM Trans. Program. Lang. Syst.*, 28(1):1–69, January 2006.

[72] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[73] M. Macedonia. The gpu enters computing's mainstream. *Computer*, 36(10):106–108, oct 2003.

[74] Virendra J. Marathe and Michael L. Scott. A qualitative survey of modern software transactional memory systems. Technical Report TR 839, University of Rochester Computer Science Dept., Jun 2004.

[75] Dale E. Martin, Timothy J. McBrayer, and Philip A. Wilsey. WARPED: A time warp simulation kernel for analysis and application development. In *HICSS '96: Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*, page 383. IEEE Computer Society, 1996.

[76] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[77] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC, pages 267–275. ACM, 1996.

[78] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.

[79] B.A. Myers, R.G. McDaniel, R.C. Miller, A.S. Ferrency, A. Faulring, B.D. Kyle, A. Mickish, A. Klimovitski, and P. Doane. The amulet environment: new models for effective user interface software development. *Software Engineering, IEEE Transactions on*, 23(6):347–365, June 1997.

[80] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[81] David Pacheco. Postmortem debugging in dynamic environments. *Communications of the ACM*, 54(12):44–51, December 2011.

[82] D. Patterson. The trouble with multi-core. *IEEE Spectrum*, 47(7):28–32, 53, jul 2010.

[83] Pin. http://www.pintool.org/.

[84] Bruno R. Preiss, Wayne M. Loucks, and D. MacIntyre. Effects of the checkpoint interval on time and space in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, 4(3):223–253, July 1994.

[85] QPT. http://pages.cs.wisc.edu/∼larus/qpt.html.

[86] Francesco Quaglia, Alessandro Pellegrini, Roberto Vitali, Sebastiano Peluso, Diego Didona, Giovanni Castellari, Valerio Gheri, Diego Cucuzzo, Stefano D'Alessio, and Tiziano Santoro. ROOT-Sim: The ROme OpTimistic Simulator - v 0.99 RC-1. `http://www.dis.uniroma1.it/~hpdcs/ROOT-Sim/`, October 2011.

[87] Francesco Quaglia and Andrea Santoro. Non-blocking checkpointing for optimistic parallel simulation: Description and an implementation. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):593–610, june 2003.

[88] T. Ragunathan and P. Krishna Reddy. Improving the performance of read-only transactions through speculation. In *Proceedings of the 5th international conference on Databases in networked information systems*, DNIS, pages 203–221. Springer-Verlag, 2007.

[89] Robert Rönngren and Rassul Ayani. Adaptive checkpointing in Time Warp. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pages 110–117. Society for Computer Simulation, July 1994.

[90] Robert Rönngren, M. Liljenstam, Rassul Ayani, and J. Montagnat. Transparent incremental state saving in Time Warp parallel discrete event simulation. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pages 70–77. IEEE Computer Society, May 1996.

[91] A. Roth and G.S. Sohi. Speculative data-driven multithreading. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 37 –348, 2001.

[92] Peter G. Sassone and D. Scott Wills. Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 7–17. IEEE Computer Society, 2004.

[93] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):18:1–18:15, August 2008.

[94] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, August 1995.

[95] G.S. Sohi and A. Roth. Speculative multithreaded processors. *IEEE Transactions on Computers*, 34(4):66–73, apr 2001.

[96] H.M. Soliman and A.S. Elmaghraby. An analytical model for hybrid checkpointing in Time Warp distributed simulation. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):947–951, october 1998.

[97] SPEEDES. `http://www.speedes.com`, 2005.

[98] Jeffrey S. Steinman. Incremental state saving in SPEEDES using C plus plus. In *Proceedings of the Winter Simulation Conference*, pages 687–696. Society for Computer Simulation, december 1993.

[99] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[100] The SCO Group, Inc. *System V Application Binary Interface*, fourth edition, March 1997.

[101] The SCO Group, Inc. *System V Application Binary Interface, Intel386 Architecture Processor Supplement*, fourth edition, March 1997.

[102] Roberto Toccaceli and Francesco Quaglia. DyMeLoR: Dynamic Memory Logger and Restorer library for optimistic simulation objects with generic memory layout. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, pages 163–172. IEEE Computer Society, 2008.

[103] Valgrind. http://valgrind.org/.

[104] Robert Wahbe, Steven Lucco, and Susan L. Graham. Practical data breakpoints: Design and implementation. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, 1993.

[105] Darrin West and Kiran Panesar. Automatic incremental state saving. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pages 78–85. IEEE Computer Society, May 1996.

[106] Jae-heon Yang and James H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9:9–1, 1994.

[107] K. Yelick, P. Hilfinger, S. Graham, D. Bonachea, J. Su, A. Kamil, K. Datta, P. Colella, and T. Wen. Parallel languages and compilers: Perspective from the titanium experience. *Int. J. High Perform. Comput. Appl.*, 21:266–290, August 2007.

# 8 Educational plan

| Type | Name | Credits | Date |
|---|---|---|---|
| A | | 12/12 | |
| | "Artificial Intelligence II" (Prof. D. Nardi) | 6 | 02-10-2012 |
| | "Elective in Computer Networks" (Prof. L. Becchetti) | 3 | 28-06-2011 |
| | "Pattern Recognition" (Prof. F. Pirri) | 3 | 23-09-2011 |
| B | | 10/10 | |
| | "Winter School on Hot Topics in Distributed Computing", La Plagne, France | 2.5 | 2011-03-25 |
| | "GII Doctoral School", Lucca, Italy | 2.5 | 2012-07-04 |
| | "Scuola Estiva di Calcolo Avanzato, Grottaferrata, Italy | 5 | 2011-09-09 |
| C | | 8/8 | |
| | "Chippy's Recovery" (Donald R. Perlis) | 0.5 | 20-01-2011 |
| | "Control, Recognition, Planification - The task function, swiss knife of the humanoid robot" (Nicolas Mansard) | 0.5 | 20-09-2011 |
| | "DISC Workshops & Tutorials" | 4 | 22-09-2011 |
| | "Markovian Agent Models with Applications" (Andrea Bobbio) | 0.5 | 06-07-2012 |
| | "Microsoft Windows Azure" (Antimo Musone, Daniele Midi) | 0.5 | 04-05-2012 |
| | "Online Generation of Kinodynamic Trajectories" (Boris Lau) | 0.5 | 06-05-2011 |
| | "Programming by Optimisation: Towards a new Paradigm for Developing High-Performance Software" (Holger Hoos) | 0.5 | 13-05-2011 |
| | "Robotics: Hephaisots reoffends" (Jean-Paul Laumond) | 0.5 | 29-04-2011 |
| | "Network Science: From Structure to Control" (Albert-László Barabási) | 0.5 | 05-07-2012 |