

Hardware Transactional Memory:

In building a hardware transactional memory, the main challenge is in building a system that provides full-speed execution for simple, reasonably-sized transactions, while providing support for rich transaction semantics (arbitrary sizes, fancy nesting, debugging, etc.) with reasonable cost and without sacrificing any other machine properties (e.g., performance isolation, virtualizability). Our first paper, identified the challenges to providing performance isolation in Hardware TM systems [1].

Our second paper extends a hardware TM with hooks to software to provide a cost-effective (in terms of hardware) means to providing richer transactional semantics, including support for pausing transactions, registering compensating actions, and atomically descheduling and waking up transactions rather than aborting them when conflicts occur [2].

Transactional Memory Workloads: One of the workloads that we've tried to parallelize with transactions is the Python interpreter [3]. While the Python programming language provides threads as part of the language, its canonical implementation (CPython) provides only limited concurrency as bytecode execution is performed while holding a "global interpreter lock." With the expectation that the bytecodes would likely be embarrassingly parallel, one of my students undertook replacing this overly conservative concurrency control mechanism with the optimistic execution and fine-grain conflict detection provided by wrapping each bytecode with a (hardware) transaction. Our lessons learned were two-fold: 1) it was remarkably easy to eliminate the false conflicts resulting from the interpreter's use of global variables, and 2) it was remarkably hard to correctly deal with the bytecodes (and therefore the transactions) that resulted in system calls and I/O, in part because they may be encapsulated in native code. In hindsight, we recognized that exposing Python's concurrency would have been much easier by programming to a hardware abstraction that speculatively executed lock-based critical sections (*i.e.*, Speculative Lock Elision (SLE)), which would have transparently handled the system call and I/O issues correctly. We believe that this conclusion likely generalizes to many of the workloads that have been executed on TM prototypes.

Using Hardware Checkpoint/Undo Support to Facilitate Reliable Compiler Optimization: This work is concerned with hardware support to make microprocessors better compiler targets, making software optimizations more effective, safer, and more reliable [4]. Given that programs tend to spend much of their time executing a small fraction of the possible paths through the program, there is significant opportunity to improve performance and reduce power consumption by optimizing the code along these hot paths. Traditional approaches to speculative compiler optimizations (*i.e.*, those that improve one path at the expense of other colder paths) are significantly more complex than their non-speculative counterparts, because they must be designed to be path specific and to generate compensation code for exits from the hot path(s). We demonstrate that the benefit of these optimizations can be achieved, or even exceeded, without this complexity by providing hardware primitives for checkpointing and software-initiated rollback. These primitives provide to the compiler the ability to isolate the program's hot path for optimization, allowing the use of non-speculative formulations of optimization passes. When a speculation invariant does not hold, the checkpoint is restored and control is transferred to a non-speculative version of the code, relieving the compiler from the responsibility of generating compensation code.

[1] "Challenges to Providing Performance Isolation in Transactional Memories," *Craig Zilles and David Flint (Workshop on Debunking, Duplicating and Deconstructing 2005)*.

[2] "Extending Hardware Transactional Memory to Support Non-busy Waiting and Non-transactional Actions," *Craig Zilles and Lee Baugh (Transact 2006)*.

[3] "Hardware Transactional Memory Support for Lightweight Dynamic Language Evolution," *Nicholas Riley and Craig Zilles (Dynamic Languages Symposium 2006)*.

[4] "On the Effectiveness of Atomic Regions for Reliable Compiler Optimization," *Naveen Neelakantam, Ravi Rajwar, Suresh Srinivas, Uma Srinivasan, and Craig Zilles (ISCA 2007)*.