Dynamic Identification of Shared Transactional Locations

Alexander Matveev  
Tel-Aviv University  
matveeva@post.tau.ac.il

Nir Shavit  
MIT and Tel-Aviv University  
shanir@csail.mit.edu

Abstract

Hardware TM systems execute user code within an \texttt{atomic\{} \texttt{}} delimiter without any instrumentation. Software transactional memory systems require complex sequences of operations to be executed on the memory locations shared by transactions, but typically not on unshared locations, even if these are accessed within the scope of a transaction. Lack of identification of such instructions introduces a large performance overhead. The problem of identifying if an instruction is accessing a location which is shared, even if these locations are declared in advance, is a dynamic runtime problem, i.e. not solvable effectively through the use of a compiler.

In the spirit of the new trend towards hardware assisted STMs (HASTMs), we show how one can add a simple hardware element, the Virtual Memory Filter (VMF), that provides dynamic identification and execution of STM functions on transactionally shared locations. The VMF will provide STMs with the simplicity of HTMs: \texttt{atomic\{} \texttt{}} code can execute “as is.” Its introduction into commercial CPUs will eliminate the need to use a compiler to transactify user code, a benefit currently claimed only by full fledged HTM systems. Our preliminary empirical evidence shows that the VMF component has virtually no performance penalty.

1. Introduction

Modern software transactional memory (STM) schemes still have a way to go on the path to being widely adopted. A major obstacle on this path is the effective application of the transactional interface to code. In a sound byte, the transactional interface is a delimiter such as \texttt{atomic\{} \texttt{\}} that wraps the instructions within the scope of a given transaction. One of the advantages of hardware transactional memory (HTM) systems, is that the code within the atomic delimiters is executed “as is”, without any need to instrument the memory accesses.

Unfortunately, STMs require instrumentation, yet none of the existing instrumentation techniques is both simple for the user and efficient in the performance it provides. In a typical application of the transactional interface to code, operations on both shared and unshared memory locations are included by the programmer within a transaction’s scope. There is no way to avoid this. Dynamic transactions, and especially unbounded ones, must include local variables, library calls, etc, and any TM implementation is required to execute both as part of the transaction’s flow. However, within this code, shared locations must be accessed in a transactional manner running specialized TM code, while unshared ones can run in a non-transactional manner, that is, with a minimal set of added TM operations. Typically, TM schemes, even the most efficient of them, introduce an overhead in transactional accesses. It is therefore crucial to identify which instructions need to be executed transactionally since the performance of the whole TM based implementation is affected by this overhead [1, 7]. This has lead researchers to show interest in finding efficient ways to dynamically separate non-shared memory accesses from shared ones [6, 7].

Compilers can help mitigate the problem to some extent [3]. In general, however, the process of identifying which instructions in the code access shared locations, and which do not, is complicated since many of these accesses are determined only at run time. During a program’s lifetime, a variable may repeatedly switch between pointing to unshared locations and shared ones. In other words, it is not obvious how to efficiently instrument code to eliminate unnecessary transactional accesses. As we show in the performance section (see Figure 4), adding even minimal instrumentation (a test consisting of a jump and a compare) to every non-shared access in a C program using the TL2 STM, resulted in unacceptable performance penalties. In Section 6 we explain how state-of-the-art TM systems currently deal with the memory identification problem.

This paper joins the recent trend of providing hardware assisted STMs (HASTMs) [27–29, 33]: building simple hardware components into processors in order to make STMs behave like HTMs, without the design costs and limitations of an HTM. We show how to effectively solve the shared access identification problem, using a simple hardware mechanism. We introduce the \texttt{virtual memory filter} (VMF), a simple hardware mechanism that can be added before the standard virtual memory translation mechanism (of virtual addresses to physical addresses) in commercial processors. The programmer will only be required to declare which locations are shared and which are unshared. We believe this is a simple requirement that will become acceptable and perhaps standard in programming languages as shared memory multiprocessors become commonplace (not just for purposes of TM performance but also as aids in GC, security, etc).

The VMF will allow any STM, Hybrid TM, or HASTM, to \textit{dynamically} identify instructions accessing shared memory and execute the appropriate transactional actions. Transactional reads and writes will not need to be declared by explicit API calls or instrumented in any way (see example in Section 2). This means that the source code will stay “clean” and can be compiled with any standard compiler (i.e. no compiler or pre/post compiler modifications are required). We believe this benefit is very important in making STM programming broadly accepted. Moreover, because shared read and write accesses will be detected dynamically, transactions will be able to easily access precompiled code (such as a library) as long as it is abortable.

Being software people, our VMF design may lack various properties that would not escape a hardware designer. However, we be-
lieve it is important to put forth such a mechanism, with the hope that computer architecture designers will pick up on the general benefits of the VMF approach and perhaps come up with a streamlined architectural element fitting modern day processors.

1.1 Leveraging Existing Virtual Memory Hardware

To give the reader a feel of our approach, we first present a solution to the identification problem that runs on today’s machines. It will use the processor’s existing hardware virtual memory (VM) mechanism. We will call this solution virtual memory STM (VMSTM). This solution is not in itself viable because of the VMs limitations, but serves to show how effective a solution using our suggested hardware VMF component might be. Here is how it works.

To define a transaction in VMSTM, the programmer will only need to declare which range of memory is transactional, perform the shared operations on that range (for example transactional allocations [4]), and place \textit{begin-transaction} and \textit{end-transaction} delimiters in the code to mark the transaction’s scope. No other code modifications are necessary!

The VMSTM system will use the virtual memory paging and OS exception mechanisms to dynamically detect access to the transactional range. The system wraps the transaction with a try/except block and executes it on a “shadow copy” of the transactional range. The shadow copy is a reserved virtual memory range of the same size as the real range whose pages are not committed. Upon first access to a page in the shadow copy range an exception is raised, the page is committed, it’s data is copied from the real range, and the execution is resumed (See Figure 1). Subsequent accesses to the same page will not generate an exception. Upon committing of the transaction, the modified pages from the shadow copy are copied to the actual memory range. Thus, the trapped STM code is only executed when the transaction begins, on the very first access to any page, and when the transaction ends. The burden of dynamic shared memory detection for every given instruction is thus transferred to the hardware VM detection mechanism, and involves no exception handling in the common case. The only performance penalty it incurs is that of a context switch due to an interrupt upon every first page access.

Unfortunately, the granularity of VMSTM is page size only. Thus, for example, if it uses the TL2 STM algorithm [5], locking will be at page granularity, which is a major drawback. We could not find in the literature on processor hardware a mechanism that we could leverage in a similar way to reduce the access granularity unless we were willing to allow code rewriting at runtime. Code rewriting is problematic since it typically cannot be applied to shared libraries, and in any case requires complex changes to the program and OS code.

1.2 The Virtual Memory Filter

Our main proposal is a simple new hardware protocol, the virtual memory filter (VMF), which solves the problem of shared memory identification at any level of granularity while requiring minimal changes to the CPU. The VMF is not limited to one particular STM algorithm: it can serve as a basis for effectively running many types of commit-time [8, 20] STM protocols. The idea is to add a small hardware filter before the existing VM translation mechanism of virtual addresses to physical addresses. This detector will identify which accesses are to the transactional (shared) range and which to the non-transactional (unshared) range. For each identified shared address, it will map it to shadow copy of the shared range and invoke an appropriate software handler. A different software handler or direct memory store operations can also be invoked for unshared locations. This will allow fine-grained memory access tracing, which in turn will allow an STM running on top of this

![Figure 1. The VMSTM executing a transaction wrapped by a try/except accessing different types of memory. First a regular access is performed. Then a first transactional access to page 2 of the shadow copy triggers a page copy from the real buffer. Then, another standard access is performed.](image1)

![Figure 2. The VMF block flow of actions for a regular (non-shared) access and a shared access. The regular accesses are not effected by the block. Transactional accesses on the other hand trigger a software handler and are mapped to a shadow copy of the shared range. The software handler invocations allow tracing of shared accesses. An address mapping to the shadow copy works on the copy and not on the real data.](image2)
OS intervention. Upon detection of memory access that requires a software handler invocation, a “jump” instruction will be simulated by changing the program counter to the start address of the software handler. The penalty for this is at most that of a pipeline flush.

1.3 Hardware

A simple way to implement the VMF idea in hardware is to add the VMF hardware filter before the virtual-to-physical memory translation and simulate the jump to the software handler by a pipeline flush. This simple hardware will be enough for our STM application. A pipeline flush per STM handler invocation is much better than instrumenting all memory access instructions inside the transaction block. But in general, the software handler invocation can be frequent in different applications resulting in frequent pipeline flush. To avoid the pipeline flush, a more complex hardware implementation is presented next, that includes a VMF prediction table and a VMF stripe cache. The prediction table is similar to the branch prediction tables used in modern CPUs. It will instruct the instruction fetching unit when the “jump” to a software handler is required and avoid the pipeline flush on correct prediction. The stripe cache will allow to implement a handler invocation once per specific memory location. For example, tracing if access was made to some location in some code block can be implemented by executing a handler on first access to this location. Next accesses to this location in this code block are not required to execute the handler.

1.4 Performance

As a preliminary test to show its efficacy, instead of fully emulating our new VMF (which will be performed for the full version), we will show that a software mechanism that emulates it in a crude way using the VM mechanism provides good performance. This emulated VMF based STM is at least as costly as any hardware implementation of VMF would be. We compared code running the the emulated VMF using TL2 to code modified using standard the TL2 STM, varying the number of threads and the amount of non-shared accesses. In both cases, the emulated VMF had nearly the same behavior as the original hand instrumented code. This is encouraging since the a real VMF hardware can perform only better.

2. The Virtual Memory STM

We begin by introducing the virtual memory STM (VMSTM), an STM with a memory identification scheme using existing hardware support for virtual memory. To use the VMSTM, a programmer reserves a transactional range in memory (this is not a special hardware buffer, just a reserved section of existing memory). Transactional allocations of shared memory and subsequent accesses will happen in this range. For example:

```c
int main() {
    VMSTM_HANDLE hVMStm = NULL;
    hVMStm = VMSTM_InitBuf(10 * PAGE_SIZE);
    StartThreads(1...N);
}
```

```c
int ThreadFunc(VMSTM_HANDLE hVMStm) {
    PVOID pStmBuf = NULL;
    VMSTM_THREAD_HANDLE hTxContext = NULL;
    hTxContext = VMSTM_OpenThreadHandle(hVMStm);
    pStmBuf = StartTx(hTxContext);
    ...}
```

The transactional range is allocated by reserving and committing the requested number of pages in virtual memory. Every thread opens a “handle” to the transactional range and executes transactions by wrapping a block of code with StartTx(hTxContext) and EndTx(hTxContext) calls. The code block itself is not modified. Opening a “handle” in some thread causes the creation of a shadow copy of the transactional range which is a reserved range of the same size but whose pages are not committed (See Figure 1). Pointer to base of the shadow copy is returned to the user when the transaction starts. Then, the transaction accesses the pointer as it would access any standard buffer. To handle these accesses correctly, we use the paging mechanism combined with the OS exception mechanism. StartEx and EndEx are macros which expand to try ... except (...) .. block and upon access violation to shadow copy the following is applied:

1. Commit Check: If access address’s page is not committed then, commit the page, initialize it by matching the page’s data from the transactional range.

2. Access Check: If access was a read, then set page access to read-only. Otherwise, set the page access to read-write.

3. Resume: Resume the execution from the instruction that caused the access violation.

This combination of paging with exception handling allows us to trace memory accesses at page granularity. We get an exception for every first access for a read or a write for every page. Therefore, the read-set and write-set granularity, and hence the lock granularity, can be only per page. For example, in our benchmarks we implemented the TL2 STM [5] algorithm on top of the VMSTM using a version-lock per page.

The benchmark of VMSTM based on TL2 showed bad results relative to standard TL2. That’s because the usage of paging and exception mechanisms which are an OS services. Using them requires interrupts to the kernel a fact introducing a high performance penalty. We tried to minimize the overall number of exceptions and paging API usage by using caching for read-set and write-set. This improved the single-threaded performance but did not give us a scalable solution. That’s because when conflicts occur the caches need to be updated and again the exception and paging mechanisms are used.

Memory trace granularity can be reduced by raising an exception for every access to the shadow copy. This however is very inefficient. Unfortunately, on current CPU and memory architectures, we did not find any other CPU element that could be leveraged in a similar way at a granularity below that of complete pages.

3. The Virtual Memory Filter

We propose adding a simple independent hardware element, the virtual memory filter (VMF) before the virtual to physical memory translation circuitry. First we will describe the general VMF architecture and then the specific implementation for our STM needs.

A general VMF hardware element architecture would be to install it before the MMU (virtual to physical memory translation unit) in order to intercept the VM space access addresses. So, it’s input and output is a VM space address. VMF logically divides
the VM space to a constant number of disjoint ranges a union of which is a whole VM space. To implement this a constant number of \([\text{base\_reg}, \text{len\_reg}]\) register pairs can be used to describe the VM ranges. For a given input address it identifies to which range it belongs and performs “actions” related to that range. Those actions can be of various types. For instance it can be updating some memory address or a register (directly without interfering with the current instruction stream), software handler invocation by pipeline flush, software or hardware signal, arithmetic manipulations of the VMF output and so on. For example, consider dividing the VM space to three disjoint ranges called the red, green and blue range (union of the three is a whole VM space). For every input address from the red range, VMF won’t do anything. For every address from the green range it will invoke a software handler (by pipeline flush) and manipulate the output. Finally, for addresses of the blue region it will increment a hardware register called \(\text{reg\_counter}\). As a result the VMF code would be as follows:

\[
\text{VMF}(\text{VM\_Addr}): \begin{align*}
1. & \quad \text{if base\_reg\_red} \leq \text{VM\_Addr} \text{ and} \\
& \quad \text{VM\_Addr} < \text{len\_reg\_red} \text{ then:} \\
& \quad \quad 1.1 \quad \text{return} \quad \text{VM\_Addr} \quad // \text{do nothing} \\
2. & \quad \text{if base\_reg\_green} \leq \text{VM\_Addr} \text{ and} \\
& \quad \text{VM\_Addr} < \text{len\_reg\_green} \text{ then:} \\
& \quad \quad 2.1 \quad \text{Flush the pipeline and} \\
& \quad \quad \text{set PC to the software handler start} \\
& \quad \quad 2.2 \quad \text{Resume execution at PC.} \\
& \quad \quad (\text{next step will execute} \\
& \quad \quad \text{after the function finished}) \\
& \quad \quad 2.3 \quad \text{return} \quad \text{VM\_Addr} / 32 \\
3. & \quad \text{else, // blue range} \\
& \quad \quad 3.1 \quad \text{reg\_counter} = \text{reg\_counter} + 1 \\
& \quad \quad 3.2 \quad \text{return} \quad \text{VM\_Addr}
\end{align*}
\]

As we can see VMF enables us to perform different actions for different VM space ranges. So, VMF can be used to profile memory accesses, control memory accesses, translate memory accesses to different addresses, perform advanced memory debugging and, in our case, used for STM purposes. In case of memory profiling VMF actions for the different ranges can be done in parallel to the instruction execution stream. Therefore VMF actions will be only to “mark for execution” and the “big actions” will be done in software by a separate threads. So VMF won’t need to interfere with the instruction stream and the profiling will be done with virtually no penalty. VMF can be made as reprogrammable chip in order to support different semantics according to program needs. Now we will describe specific implementation of the VMF for the STM algorithms.

The idea behind the VMF element for STM is very simple:

1. **Filter:** If a current memory access address is inside the transactional range then continue to next step. Otherwise do nothing.
2. **Handle:** Invoke a software handler, passing it the memory access address.
3. **Resume:** Resume the interrupted memory access instruction.

This may seem the same as raising an exception for every memory access to the transactional range. It is not. The handle step is executed only on the first access for read or write for every section (of a predecided granularity) of the transactional range. It is like VMSTM, just for sections smaller than a page size. In addition, these three steps do not need to generate an interrupt (like with the exception handler) and can be highly optimized, for example, by using a small cache.

Now we will introduce a detailed description of how the VMF block combines with an STM. Suppose we have a commit-time STM algorithm: one that constructs the read-set and write-set during a transaction’s execution and then uses this information to perform the commit. For example, one can use a commit-time version of the TL2 STM. We will use the VMF block to dynamically construct the read-set and write-set of an STM running at granularity \(\text{block\_size}\) on a buffer of size \(\text{stm\_buf\_size}\) in the following way:

1. **Open a Handle to the Transactional Range:** First a thread will open a handle to the transactional range. This action will create the shadow copy (as in VMSTM) and additionally will allocate an array \(\text{flags}\) of size \(\text{blocks\_num} = \text{stm\_buf\_size}/\text{block\_size}\). The \(\text{flags}\) array entry contains the flags for block \(i\) in the shadow copy.
2. **Start the Transaction:** Initialize the VMF block context to the given transactional range handle context. This will tell the VMF block where the transactional range, the shadow copy, and the \(\text{flags}\) array are located. In addition, execute the STM’s specific start code.
3. **Run Through a Virtual Execution:** During the execution, every access to an address inside the transactional range will trigger the VMF block. VMF block will do the following:

   (a) **Check the Cache:** The cache will store for every accessed block in the transactional buffer a pair of \([\text{block\_address}, \text{access\_type}]\). The \text{block\_address} is calculated by performing shift right of access\_address value by \(\log(\text{block\_size})\). The access\_type can be a read or write. If current, the \([\text{block\_address}, \text{access\_type}]\) pair will already be in the cache so go to step (c). Otherwise, go to step (b). This added cache is not necessary algorithmically and is added to optimize transaction performance.

   (b) **Execute the Software Handler:** Invoke the software handler (registered by the user). The software handler will check if access\_address’s page is committed in the shadow copy and commit it if required. Then it will update the \(\text{flags}\) of block accessed: \(\text{flags}[\text{access\_address - real\_buffer\_base}]/\text{block\_size}\). The \(\text{flags}\) can hold the information about the block’s page commit status, read access status, write access status and more. In addition, the software handler will execute the STM’s specific code, which can indicate transaction failure. In this case the software abort handler will be executed (registered by user).

   (c) **Translate the Address:** Translate the access address to a matching address in the shadow copy: \(\text{shadow\_copy\_address} = \text{shadow\_copy\_base} + \text{access\_address} - \text{real\_buffer\_base}\). Perform the instruction action (read or write) from/to \text{shadow\_copy\_address}. It is important to note that the instruction itself is being cheated, it “thinks” it accessed the real buffer but actually only the shadow copy was accessed. This cheating is achieved by putting the VMF block before virtual-to-physical address translation circuitry. The behavior is transparent for the running code. In some way it resembles how virtual memory works.

4. **End the Transaction:** Read the \(\text{flags}\) array to determine the transaction’s read-set and write-set. Execute the STM’s commit code given those read/write sets. If the transaction fails, execute the abort handler (user defined).

From the described algorithm we can see that the VMF block performs simple arithmetic based on a constant number of parameters: the real buffer base address, the shadow copy base address, the block size value and so on. All these can be implemented as registers in the VMF block, which will be initialized when each transaction starts. To support these calculations, the VMF block requires only a couple of registers. In addition, it needs to invoke a soft-
ware handler. This can be done by simulating a jump to a function: the program counter is changed to the software handler start line while discarding currently executed instructions in the pipeline. In other words, execute a pipeline flush. So the maximum performance overhead per handler invocation will be a pipeline flush.

3.1 Handling Unshared Locations

The so far described VMF algorithm does nothing for accesses to the unshared range. But locations in unshared ranges can be filtered and handled in the same fashion as shared ones. This is useful, for example, in STM algorithms, if one wants to allow rollback of local operations within a transaction that spans a block that is not a complete method call (typically transactions that span method calls will be aborted by popping the stack so all unshared accesses will be undone immediately and there is no need for a rollback mechanism). For example, we can execute a backup operation for non-shared accesses to more easily support transaction rollback. In this example backup operation does not require a pipeline flush only that a backup will be done before the new value written. Therefore, the handling of un-shared locations can be very efficient.

Suppose we have a thread for which we want to filter and handle a un-shared locations. First we would tell VMF about the un-shared range by initializing a [base_non_shared_reg, len_non_shared_reg] registers pair. Second, we would define the actions for this range. This actions would be added to the above described algorithm’s step of “Run Through a Virtual Execution” and is defined as follows: For every write access to the non-shared range:

1. **Check the non-shared range Cache:** The cache will store for every accessed block in the non-shared range a [block_address]. The block_address is calculated by performing shift right of access_address value by log(block_size). If current, the [block_address] will already be in the cache so go to step (c). Otherwise, go to step (b). This added cache is not necessary algorithmically and is added to optimize the transaction performance. Also, one can make one cache for shared and un-shared range or one for every range.

2. **Perform the backup and execute the write:** In order to perform the backup we would store for the un-shared range a shadow un-shared range of same size in the VM space. So, this step would generate a memory write operation of the accessed block to the shadow non-shared range in the respective place as in the non-shared range. This memory write will be generated before the current memory write. Also, the backup scheme can be implemented in couple of ways. We can make the backup to the shadow range and work on the non-shared range or we can copy the block from the non-shared range to the shadow and work on the shadow range. In case one, if transaction fails we need to restore the backup data but for the commit we only discard the backup. In case two, if transaction succeed we need to copy the shadow updated places to the non-shared range, but for failure we only discard the shadow new values.

3.2 Optimizations

We can optimize the VMF algorithm in a number of ways. As noted earlier, to optimize the process of transactional access by the VMF, we can add a small cache to the VMF block. This cache will hold the [block_address, access type] pairs.

Another optimization has to do with the flag array. On commit, when the STM determines the read and write sets, it will scan the flags array. The size of the flags array is the number of blocks and a block is of granularity size. Therefore, number of blocks can be large. For example, a real buffer of size 20 * 4096 with a granularity of 32 bytes has 2560 blocks. To avoid scanning the entire array, the software handler invoked by the VMF can store the accessed blocks numbers in a software array or list. In our hand crafted emulation of VMFSM using the TL2 algorithm, we succeeded to perform the addition to the list in O(1) (no need to check if item already exists).

Another optimization can reduce page copying. If a transaction aborts, the shadow copy’s committed pages can remain committed through the next execution attempt. Most likely the transaction will access the same pages as it did before. Upon a successful transaction commit, one can un-commit and fully free the shadow copy.

In summary, the VMF hardware is a simple independent block of circuitry that can be added to any current CPU architecture and allow support of efficient dynamic memory traces. It allows dynamic detection of transactional accesses and construction of read and write sets without performing any explicit STM API calls or code and compiler modifications.

4. VMF Hardware Design

The VMF hardware allows the programmer to apply special behavior to a predefined memory range. For this, a predefined software handler is executed upon every access to the defined memory range. The software handler implements the special behavior that is associated with the memory range where the hardware implements the dynamic triggering of this software handler. By placing the VMF memory filter before the virtual to physical translation mechanism (MMU) accesses to the special memory ranges can be identified. Upon identification the associated software handler can be executed by flushing the current instructions in the CPU pipeline and jumping to the software handler start address. In general, performing the pipeline flush only for dynamically identified accesses to the memory range is more efficient than instrumenting every memory access in the program. Still if the dynamic identification
is frequent, the pipeline flush may harm the overall performance. To overcome this problem the VMF prediction table and the VMF stripe cache can be added to the hardware mechanism. They will tell the fetching unit when the “jump” to a software handler is required.

The simple form of a software handler triggering pattern is when it is triggered for every access to a predefined memory range. Sometimes when the code accesses an object inside a predefined memory range an invocation is required only for the first access to the object. The first access makes some special handling and subsequent accesses can proceed freely. Therefore the requirement is to execute a software handler once per specific memory location. For this purpose a VMF stripe cache is implemented. It will cache the memory location on a first triggering and next accesses to it will not result in an invocation.

4.1 VMF memory filter
The VMF memory filter is placed before the virtual to physical translation memory unit (MMU) to identify memory accesses to the special predefined memory ranges. To define a memory range a start address and a length must be specified. Therefore the filter can be implemented by maintaining a table of this [start address, length] pairs defining multiple memory ranges. To avoid this tables another way can be used. Special color bits can be added for every page in the page table. In this case, the special memory range will be defined as the set of pages having the same value for the color bits and the number of bits determine the number of possible different memory ranges.

4.2 VMF Prediction Table
The simple implementation of software handler invocation is a pipeline flush. This can be avoided by adding a VMF prediction table in a similar way the branch prediction tables used in modern CPUs. First time the memory access instruction will trigger the software handler of some memory range, the instruction’s address and the software handler executed, will be recorded in the VMF prediction table and the pipeline will be flushed. Next time this instruction will be fetched, the recorded entry in the prediction table will force the fetching mechanism to jump to the software handler instructions avoiding the pipeline flush.

To record the instruction that resulted in a software handler invocation the table entry will have:
1. Instruction’s address (30 bits)
2. Software handler’s address (30 bits)

To be simple every entry can be predicted as always jumping to the software handler. In most of the cases the behavior of the instruction will be the same so the prediction will work. Still there can be instructions having an altering behavior causing them sometimes to trigger the VMF and sometimes not. A more complex prediction mechanism can be used by adding a prediction state machine like in a branch prediction tables.

4.3 VMF Stripe Cache
To support one software handler invocation per memory location a VMF stripe cache is required with the VMF memory filter. The memory is logically divided to stripes of constant size. The stripes are similar to pages just that they are much smaller than pages. Upon first access to a stripe it will be recorded (the base address of the stripe) in the stripe cache and the software handler will be invoked. Next accesses to the stripe will see the stripe in the cache and will not result in an invocation.

An entry in the table will hold a stripe’s start address (30 bits). The programmer will first reset this table and then execute some block of code which will result in “once per location/stripe” invocations. Then the table will be reset and some other or same block of code can be executed to perform this invocations again.

To summarize all the components together let us consider an example of a code block that accesses protected objects. We want that on every first access to a protected object a security function will be executed. The code block accesses couple of objects where every object is a structure that has 10 fields. The code block accesses all of the 10 fields one after another. Therefore on access to the first field of the object the security function must be executed. Next 9 field accesses will proceed normally. Now assume the code block is executed multiple times. The first execution of the code block will populate the VMF prediction table with the instructions that make first access to every protected object. As a result, next executions of the code block will have correct prediction and no pipeline flush. The VMF stripe cache will be reset on a start of every code block execution allowing to invoke the software handler once per protected object.

The combination of the VMF memory filter, VMF prediction table and the VMF Stripe Cache allows to implement dynamic execution of software handlers for specific memory ranges without any performance penalty to the pipeline execution. The execution will flow without interruption after the VMF prediction table and the VMF stripe cache will be saturated.

5. Performance
We present here a comparison of the TL2 algorithm [5] running under the VMSTM and VMFSTM algorithms, to state-of-the-art STM algorithms. The data structure we used for testing is a standard skiplist.

The skiplist [23] is a probabilistic structure which behaves like a balanced tree. Our implementation is derived from LibLTX that includes the original Fraser and Harris lockfree-lib package [10]. It exposes the standard put, remove, and lookup API functions. We began with a benchmark to prove to ourselves that the dynamic memory identification problem is indeed a problem, that is, that testing dynamically, in software, if a given memory location is shared or not, is not an acceptable solution (at least not in C programs). To this end, we ran a benchmark comparing versions of TL2 in which we left the non-shared accesses un-instrumented (TL2 NS NI) versus one in which the non-shared accesses are instrumented (TL2 NS I). A TL2 instrumented access starts by checking if the accessed location is shared, and so incurs a jump and a comparison for non-shared locations. Figure 4 shows the results of our benchmark. The skiplist transactions always have about 1000 shared operations and we vary the number of non-shared ones from 100 to 10,000. We can see that when the ratio of shared to unshared operations is the same, TL2 with the non-shared section instrumented is three times slower than its non-instrumented counter-part.

Having established that there indeed is a problem, we conducted a set of benchmarks to compare the proposed solutions. The general form of tests we conducted declares a transactional range, allocates the skiplist data structure, and spawns a given number of threads to operate on that range. Every thread loops until it is signaled to stop. In every loop iteration we randomly choose between the put, remove, and lookup operations. After a given time all the threads are signaled to stop, and the total number of operations performed by them is calculated.

For our experiments we used Intel™ Core 2 Quad Q6600 2 x 4MB L2 Cache processor running Windows Vista™.

The benchmarked algorithms included:

**TL2-Page**: This is an implementation of the TL2 STM algorithm running on a stripped-range. The stripe size is the page size...
VMSTM: An implementation of TL2 on top of VMSTM, which uses VM paging and exception handling to perform the read-set and write-set construction. It runs the TL2 on page granularity. In addition, it also has read and write set optimizations for reducing the number of page allocations and searching in O(1).

TL2: The same as TL2 Page with a fine-granularity. The granularity of this algorithm is the skiplist node size.

Emulated VMFTL2: An emulation of the VMFTL2 scheme for running the TL2 STM. Instead of a cycle by cycle simulation of the VMF, we created a hand crafted version of the VMFTL2 in software, a version that is at least as expensive as a VMF hardware component would be. We hand-instrumented the code to make every transactional access perform a function call (a software handler call) and to perform the translation. Performing the translation and function call for every transactional access has a higher performance penalty then doing it in real hardware. Unlike in the VMF to which one can add a cache so the software handler is not always called, here the handler is always called.

In Figure 5 we present a non-shared access impact benchmark of VMFTL2 versus TL2 with non-shared accesses not instrumented. As expected, they have the same results because VMFTL2 has no overhead for non-shared accesses.

In Figure 6 we present two skip list benchmarks performed using two different key ranges and two sets of operation distributions. The key range of 2^{12} = 4K generators a small structure while a range of 2^{24} = 16384 (large) keys creates a larger skip-list, imposing larger transaction size for the set operations. The different operation distributions represent two type of workloads, one dominated by reads (5% puts, 5% deletes, and 90% gets) and in the other (30% puts, 30% deletes, and 40% gets) most operations are writes.

As we can see, the Emulated-VMFTL2, TL2, and TL2-Page algorithm’s performance improve as the number of threads increases. For more than 4 threads the performance degrades because contention increases with no added throughput. We can see that VMFTL2 and TL2 performance are nearly the same. In addition, as smaller the data structure and as more contention/threads is added the more impact on performance it does. Therefore on small skip list the degradation is much faster than on the large one. From the same reason the difference between write work-load and read work-load for smaller data structure is higher than for the larger one.

We initially expected that the performance of VMSTM and TL2-Page would be the same. Unfortunately, VMSTM is significantly slower. To understand why we conducted several benchmarks which we do not described here, and discovered that the cause is the overhead of the exception handler and kernel API calls as VMSTM makes user-kernel mode transfers which involve interrupts and OS code running. Unlike TL2-Page, VMSTM has at least one context-switch for every transaction executed.

In contrast to the gap between VMSTM and TL2-Page, there is no gap between Emulated-VMFTL2 and TL2: they perform nearly the same. This is encouraging as Emulated-VMFTL2 simulates the VMF block using a costly function call, implying that VMF in hardware would behave as well as, if not better than, the standard TL2 hand-instrumented algorithm.

### 6. The State-of-the-Art

Let us understand how the state-of-the-art TM systems deal with code instrumentation to support a transactional interface. These include (1) hardware transactional memories HTM that support execution of code in hardware without instrumentation, (2) library based STMs such as [5, 6, 9–21] that require the programmer to instrument the code by hand, changing load and store instructions into transactional loads and stores, using on-the-fly tests to check whether a memory word or block should be used by the transaction, or by defining rules on the language usage, and (3) compiler supported STMs [8, 35] that use a compiler to perform code instrumentation.

HTM implementations eliminate the need to instrument code because they perform detection of shared and unshared locations efficiently in hardware during the execution. However, hardware transactions, at least in the foreseeable future, will most likely be limited in their size and their semantics. Thus, many researchers are focusing on hybrid (HyTM) [24–26, 29, 30, 32, 34–38] systems, and more recently on hardware supported STMs (HSSTM) [31, 33], as the right way to provide unbounded size transactions with limited hardware support. HyTMs attempt to run the transaction fully in hardware and default to software if the transaction overflows, while HSSTMs provide partial hardware support for a system that is executed fully in software. The mechanism we propose here is a novel hardware support element that will solve the memory identification problem in HyTM and HSSTMs. It will allow, to a large extent, to eliminate the need to use a compiler to transactify the memory access instructions in user code, a benefit currently available only on full fledged HTM systems.

There is a class of STMs, including many existing experimental STMs, that require the programmer instrument the code by hand. This has the advantage of the programmer knowing to which in-
Figure 6. The top two graphs show the throughput of skip-list using VMFTL2 and TL2 with 5% puts and 5% deletes and 30% puts, 30% deletes. The bottom two graphs show this for VMSTM and TL2. We can see that VMFTL2 and TL2 behave nearly the same, while the VMSTM incurs significant overheads relative to the TL2.

7. Conclusions

We showed a novel VMF mechanism that automatically detects all types of shared transactional accesses and separates them from non-transactional ones. It simplifies programming in that transactional code can be executed in an unmodified manner, effectively removing the need for a compiler to transactify the code in order to apply the transactional interface. The scheme is not limited to one particular STM algorithm: it can serve as a basis for effectively running many types of commit-time [8, 20] STM protocols.

We note that the mechanism is not limited to transactions, and expect that in the future it will find applications in the detection of shared vs. non-shared memory in other multiprocessor applications as well. For example, debuggers can use it to detect shared memory accesses, profiling applications can use it to detect hot-spots, security mechanisms can use it to apply security policies at a fine granularity on memory regions [22] and so on.

References


