# Dynamically Filtering Thread-Local Variables in Lazy-Lazy Hardware Transactional Memory

Sutirtha Sanyal[1], Sourav Roy[2], Adrian Cristal[1], Osman S. Unsal[1], Mateo Valero[1]
[1]Barcelona Super Computing Center, Barcelona;[2]Freescale Semiconductor India Pvt. Ltd.
[1]{sutirtha.sanyal,adrian.cristal,osman.unsal,mateo.valero}@bsc.es;[2]sourav.roy@freescale.com

***Abstract*—Transactional Memory (TM) is an emerging technology which promises to make parallel programming easier. However, to be efficient, underlying TM system should protect only true shared data and leave thread-local data out of the transaction. This speed-up the commit phase of the transaction which is a bottleneck for a lazily versioned HTM.**

**This paper proposes a scheme in the context of a lazy-lazy (lazy conflict detection and lazy data versioning) Hardware Transactional Memory (HTM) system to identify dynamically variables which are local to a thread and exclude them from the commitset of the transaction. Our proposal covers sharing of *both* stack and heap but also filters out local accesses to both of them. We also propose, in the same scheme, to identify local variables for which versioning need not be maintained.**

**For evaluation, we have implemented a lazy-lazy model of HTM in line with the conventional and the scalable version of the TCC in a full system simulator. For operating system, we have modified the Linux kernel. We got an average speed-up of 31% for the conventional TCC, on applications from the STAMP benchmark suite. For the scalable TCC we got an average speed-up of 16%. Also, we found that on average 99% of the local variables can be safely omitted when recording their old values to handle aborts.**

***Index Terms*: Hardware Transactional Memory, Virtual Memory Management, Dynamic Heap Separation, Translation Lookaside Buffer(TLB).**

## I. Introduction

Recent emergence of multi-core processors has generated interest in writing concurrent programs for mainstream applications. With multiple independent cores programmer can truly exploit the Thread Level Parallelism (TLP). Nevertheless writing parallel programs for shared memory multiprocessors is difficult. Specifically maintaining the atomicity property of a code across multiple threads with acceptable scalability is a challenge. The classical lock-based way of handling this issue is error-prone, non-composable and often non-scalable.

The Transactional Memory[6], [4], [11], [7], [5] is an emerging concept where programmer wraps a portion of code in what is known as a transaction. It is then the responsibility of the TM framework to ensure the Atomicity and the Isolation property of that code.

However, often transactions are defined on a large piece of code containing numerous thread-local[1] variables along with a few true shared variables. Most of the time, accesses to the thread-private stack are local (though some part of it can be shared as well among other threads). Additionally some of the dynamically allocated data structures are local as well. We can safely put all these local references out of the scope of the transaction. In this article, we discuss how this can be implemented in a HTM and we show that our proposal greatly reduces the commit overhead for a lazy-lazy (i.e. lazy conflict detection and lazy data versioning) HTM[6], [4]. Also we will show how it eliminates, on average, versioning requirement for 99% of all local variables.

Specifically our major contributions are:

- We have proposed a scheme to separate dynamically shared and local data for a lazy-lazy Hardware Transactional Memory system. It can handle several sharing semantics *on-line*. Specifically we have covered cases where *Stack, Heap and Globally-declared* variables can be shared among threads. Local references made inside the transaction are committed only till the L1 level. Therefore by saving those additional trips to L2 which is approximately 10x times slower than a L1 access, we achieve speed-up. Simulation results revealed that, speed-up is possible for *both* the bus-based conventional TCC[6] and the Scalable-TCC[4].
- We have also proposed in the same scheme how to identify dynamically local variables for which it is possible to forgo versioning without affecting correctness. Our results show that it is possible to omit versioning for 99% of the local variables on average.
- We clearly articulate the implementation of our proposed algorithm using an open-source Operating System, Linux and one exemplary processor architecture, Alpha 21264.

## II. Related Work

There have been several proposals for Transactional Memory (TM) that expand on the early work by Moss and Herlihy[7]. An approach to speed-up the existing HTM is taken here by eliminating all local references from becoming the bottleneck. To achieve this, we have proposed a novel technique which automatically separates all local variables from the true-shared data within a transaction and also establishes a new way of looking into the versioning of variables. The end result is having a significant speed-up and savings in the power consumption at the same time. We have evaluated our proposal for two different flavors of lazy-lazy HTM, a) Scalable-TCC[4] and b) Conventional Bus-based TCC[6]. These HTMs unlike Log-TM[11] or One-TM[2] do not support overflow by design, but still represent an important class of HTM since they are

[1]In this article terms "thread-local" and "local" will be used interchangeably.

```
int global_int = 0;
void *f(void *arg)
{
int x = 0, i;
int *y = (int*)arg;
int *z = (int*)local_malloc(sizeof(int));
*z=0;
for (i=0;i<5;i++)
 {
  begin_tran();
  {
      x++;//PRIVATE
      (*y)++;//SHARED
      (*z)++;//PRIVATE
      global_int++;//SHARED
      *y += g(y);
  }
  end_tran();
 }..
```

```
int g(int *x)
{
   int k;
   k=(*x)+1;//PRIVATE
   *x=*x+1;//SHARED
   return k;
}

 int main()
 {
  int i;
  int p=0;// SHARED STACK VAR.
  for(i=0;i<4;i++)
      pthread_create(&th[i],NULL,f,(void*)&p);
```

(a) An example code showing optimizations

```
int * global_int ;
void *f(void *arg)
{
int x,l;
int *y = (int*)arg;
int *z = (int*)local_malloc(sizeof(int));
*z=0;
for (i=0;i<5;i++)
 {
  begin_tran();
  {
      global_int = &x; //THIS SHARING IS NOT PERMITTED.
      (*global_int) += 1;
  }
  end_tran();
 }...
```

(b) An example code showing unsupported sharing

Fig. 1. Example codes showing permitted and forbidden sharing

more fault-tolerant and robust[4]. Scalable-TCC is the only HTM proposed which implements TM in a Distributed Shared Directory large-scale multiprocessor system. A related work was carried out by Yen et al. in [13] for eager-eager (eager conflict detection and eager version management) class of HTM such as Log-TM.

An alternative approach to achieve the same goal would be the introduction of special load and store instructions which replaces normal load store while addressing the shared data[9]. This will however require a significant modification in the compiler and also cause the modified ISA to be carried on as a legacy in the future generation of processors.

## III. Motivating Examples

In this section we will describe the programming model which we presume for these optimizations. We assume that a programmer knows data structures in the program which are going to be thread local. Our proposal gives the programmer a means to propagate this information to the micro-architectural level. We require the programmer to only use a different memory allocator (section IV-B) for such thread local data structures, named *local_malloc()*.

Programmer can share data using either a) One shared heap containing all global variables which are to be accessed by all threads b) Globally declared shared variables or c) Stack variables from stack frames which are *not* executing the transaction. During execution of the program a thread also needs to allocate heap data which may eventually be shared or may remain local. Depending on the nature of the data, programmer can use either regular malloc or local_malloc() anywhere within a thread, both inside or outside of the transaction[14]. In case a programmer is using a third party library function which returns value using a heap (like strdup call in C), it is safe to use regular malloc. This is necessary since it is not known how the returned data will be used at the return site of a user program.

An example pseudo-code highlighting a typical parallel program using posix threads is shown in Figure 1(a). In this example we have commented out accesses which are detected by our algorithm as local and removed from the commit set of the transaction. Variable "p" in Figure 1(a) will be correctly treated as shared by our algorithm and thus we have the support for sharing of stack.

However we are precluding the case of sharing stack variables of a thread which is running the transaction to other threads. As shown in Figure 1(b), in this program every thread is trying to share a variable x, which resides in the *same* frame where the transaction is executing, by pulling a global pointer to point to its own stack frame. This is possible in case of unmanaged languages such as C. However output is dependent on how the individual thread is scheduled and because of this reason this is not an efficient way of programming and can sometime produce incorrect result.

## IV. Overview of the Proposed Scheme

Before presenting the actual algorithm in section V, we first discuss how to detect and filter out stack variables. Then we will explain the key advantages in the method we use to detect the "local" property of the heap. In the next sub-section we will also show how to do that using Linux kernel as an example. Hardware additions required to support the proposed technique will be presented in the next section. Then we will discuss how to even identify dynamically local variables which do not require versioning and show that it is possible to bypass versioning overhead for 99% of the local variables on average. Finally we will quantify the power impact of our proposed hardware changes.

### A. Filtering Local Stack Access

Following a traditional stack implementation as shown in Figure 2, all local variables which are accessed within a transaction need not be added to either the readset or writeset of the transaction. However they need to be tracked for speculative writes because during commit they need to be filtered out from the commit set. Also, their old value must be restored in case of an Abort. Although restoration is necessary only for some local variables as we will show in section IV-D.

To dynamically detect and filter out stack variables, we use two Internal Processor Registers namely Frame Pointer (FP) and Stack Pointer (SP). All local stack variables for the currently executing frame are located between the frame pointer (FP) and the stack pointer (SP). We check the referenced address against these two registers to determine if it is falling within the current stack frame or not. This check is performed as stack frames are getting expanded (or retracted). Once a

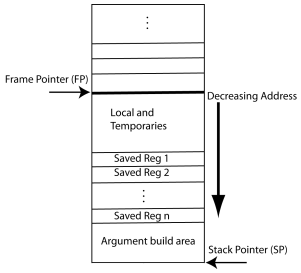



Fig. 2. A Typical Stack Organization

variable is detected as a stack variable, a special marker bit will be set in the corresponding cache line in L1 (detailed in Section IV-C and V).

### B. Filtering Local Heap Access

Apart from local stack accesses, there is another important class of local variables which can also be filtered out from the commit set. In most of the multi-threaded applications, programmer has a-priori knowledge about some data structures which reside in the heap but used only locally. As mentioned in section III, a dual version of the malloc, named *local_malloc()* is introduced to allocate the local portion of the heap. Similar to local_malloc, for freeing up the local heap, we are proposing *local_free()*.

The key technique to determine the "local" nature of the heap is to set a special bit in the virtual memory protection bit field to denote the local property of the page. However, we detect the local property of the heap as opposed to[13], which detects the shared nature of the heap. Our approach has the obvious advantage that if in any case programmer is not sure about what kind of heap should be used then default is to fall back to the shared heap allocated using regular malloc. That guarantees atomicity will be preserved.

*1) Implementation of Local Heap Separation Scheme:* In this section we will describe briefly how actually to implement the proposed local_malloc using Linux kernel 2.6.13. Our modifications are exclusively made in the OS kernel. In library we just provide two different malloc interfaces which ultimately use two different versions of the system call. For our experiment, we used "dlmalloc"[10].

To avoid duplicating system call, we have modified the existing system call *brk* (The other system call "mmap" can be similarly modified) to accept a new flag. This system call is responsible for actually going into the kernel mode and allocate pages in the virtual memory space. This flag, named *local*, controls the sharing property of the pages allocated by brk. An invocation of brk eventually gets into the core of the virtual memory allocator named *do_brk()*. Therefore we propagate the local flag till the *do_brk()* by changing its interface and implementation too. Next depending on the value of the local flag we set a special bit in the protection bit field of the allocated virtual pages. This is defined as the 21th bit in the protection bit field and is denoted as _PAGE_LOCAL. We have chosen this bit position because it is spared by the Linux kernel for future software modifications. This bit gets set if the local flag in *brk* is set to be true, which in turn gets set if user uses *local_malloc*. In case of regular malloc this bit is turned off. Now since the memory manager runs on a demand based paging scheme, physical page allocation is deferred until the last possible moment. Whenever any virtual address belonging to the local heap segment is first accessed, a TLB miss and a subsequent page fault are generated. A free page is ultimately allocated by *do_anonymous_page()* which in turn calls *mk_pte()* with the protection bits. Since earlier we have set _PAGE_LOCAL flag to be 1 in the *vm_page_prot* field in the *vm_area_struct*, this bit gets directly copied into the protection bit field of the page table entry by *mk_pte*.

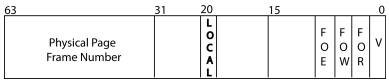



Fig. 3. Definition of new flag in kernel and new TLB entry structure

Once the physical page is allocated, any subsequent access to the physical page will bring the physical page protection flags up into the data TLB of the processor while doing the virtual to physical translation by looking up the corresponding page table entry. To store this information in TLB, we have used an available bit in the TLB entry. For the Alpha, a typical TLB entry structure (which is exactly same as the page table entry structure) is shown in Figure 3. As shown in the figure, the same 21th bit position (denoted as LOCAL) is used to hold the locality information of the page in the TLB entry.

In effect we have successfully shifted the value of the LOCAL flag from the virtual paging domain of an operating system to the individual processors TLB entry.

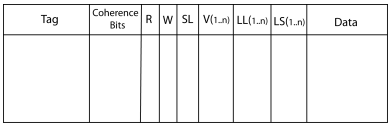
### C. Modifications to the Processor Hardware

Figure 4(a) illustrates our modifications to the data cache. Two bits, R and W, are kept as proposed in [6] to denote if a cache-line belongs to the readset or writeset (or both) of a transaction. We propose an additional bit to denote the locality of the data. We have augmented each cache line to hold a special bit called *Speculative Local* (SL). If this bit is set, then the line is treated as local and filtered out from the executing transactions commitset.

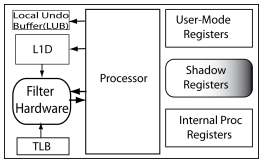
Here our assumption is that shared and local data will never share the same cache line. That is true because shared and local stack variables belong to different frames. Shared and local heaps are also allocated at page level granularity. Therefore only one SL bit per line is sufficient.

Apart from the SL bit, we are also proposing certain bits *per word in every line* to differentiate between local variables which require versioning and which do not require versioning. The *local load* (LL) bit denotes if the local variable has been read. The *local Store* (LS) bit denotes if that particular local variable has been modified. Valid bits are also kept per word to handle invalidation on a case by case basis during an abort.

Here the assumption is that all variables will be *at least padded to occupy one word* which is 16 bit in the specific case of Alpha. If this assumption is not satisfied by the compiler, then for LL and LS bits we will need even finer granularity at the level of bytes. On the other hand, if variables are *padded*




(a) A set of bits (SL) added in L1D




(b) Processor hardware modifications

Fig. 4. Proposed changes to the $\mu$-architecture

*to occupy the entire cache line*, then only R, W and SL bits are sufficient. The fundamental algorithm presented in the next section is however independent of this issue. In section IV-E we estimate the power increment of L1 due to these additional bits.

With LL and LS bits present, SL bit may seem unnecessary. However we are proposing this one bit to speed up the preparation of commit packet. During commit, any line where SL bit is set is discarded from commit. Same could have been done by checking all LL and LS bits together but with increased complexity. LL and LS bits have other use as will be discussed later.

Figure 4(b) shows the addition in the processor hardware. A standard shadow copy of the register file is needed to checkpoint the execution state when a transaction starts. It should be noted that shadow copy is required only for the software-visible register file set which are accessible in user-mode because OS kernel is assumed to be transparent with respect to transactions. Therefore shadow copy includes all general purpose integer registers, floating point registers, temporary registers, stack pointer, frame pointer and program counter. Apart from these shadow registers, our proposal does not require any other addition to the register file set. A hardware module (shown as "Filter Hardware" in the figure) is added per processor which signals processor whether a transactional load or a transactional store is happening to a local address. A small fully-associative buffer named "Local Undo Buffer" is added to hold old values of a few local cache lines of specific nature. Exact use of this buffer is detailed in the next section and we quantify a size of this buffer too. Structure of this buffer is similar to L1 except that it requires *only one* bit for the entire buffer. Because all entries in it are either valid or invalid.

### D. Local Undo Buffer

A small buffer similar to the victim cache[8] is proposed to hold old data for a few local variables. This is to ensure the correctness in case of an abort. Note that in the common case local variables do not even require preservation of the old value. For example consider the following pseudo-code shown in Figure 5. Within the transaction, x and y are explicitly written first before their values are used. *This is the common case where a local stack or local heap will be first assigned a value inside the transaction depending on the present values of the shared variables.* Then transaction will use those local variables in further computations. Finally results of those computations will be written back into shared variables. By wrapping the entire cycle inside a transaction, atomicity and isolation are accomplished. However there are some local variables whose value will be read first inside the transaction before being written. The variable i in the figure 5 corresponds to those kinds of variables. It is used as a loop-index variable and its initial value is assigned outside of the transaction. Therefore it is first read inside the transaction and then updated.

```
int f( int a)
{
int l;
...
}

int i = 0;
int x, z=50, u;
int * y= local_malloc(sizeof(int));
while (i<1000)
begin_tran()
{
x= sh_i;          ->  ldah t0,0(gp)
                      ldl   t0,-32652(t0)
                      stl   t0,16(fp)
//sh_i is a globally
declared shared variable
x=x+1;
u=f(z);
*y = // populate local heap by some value;
/* For x , *y, z, u, l old values need not be preserved */
//Do work;
                      ldl   t0,24(fp)
 i  = i +1;       ->  addl  t0,0x1,t0
                      stl   t0,24(fp)
 //Old value of i must be preserved in case of abort.
i= i + 1;
}
end_tran()
```

Fig. 5. Example code showing version-free local variables

Let us further assume a case where the transaction in the while loop gets committed first time but in the 2nd iteration gets aborted after speculatively modifying i to 3. Now with our proposal, values for all local variables such as x,*y and i are committed only till the L1 level. On Abort, in the second iteration, all speculative cache lines including the line holding the value of i are invalidated. Therefore we need a small buffer to store the old value for local variables such as i *only* because in case of an abort we must at least restore its old value. Otherwise after abort when execution re-starts, value of i will be read from memory (it will cause a L1 miss because of invalidation) as 0. Whereas the correct value should be 2. Other local variables such as x and y can be ignored because they get written first. In the figure 5, assembly output corresponding to the two classes of local variables is also shown. Variable x does not require any versioning because the type of the *first* memory request to it is *store* (The instruction "stl" is highlighted). Similar assembly output could be obtained for variable y too. On the other hand variable i will require versioning as mentioned earlier since the type of the first memory request to it is *load* (The instruction "ldl" is highlighted). We use this fact to classify these two kind of local variables.

It should be noted that the variable x is loaded and incremented immediately in the second line after the first store.

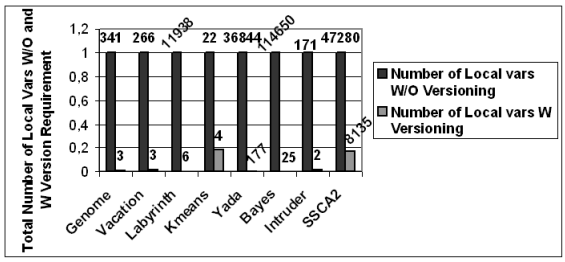



Fig. 6. Local Variables without and with requiring versioning

However this load need not be stored in the LUB. Also note that after the initial increment to i, it is incremented again. The value obtained in the first load should be backed-up in the LUB to be used in case of an abort. The value obtained in the second load should come from L1 and should *not* be stored in LUB since it does not represent the old version anymore. Variables z and u do not require any versioning as well. Variable u is analogous to x and re-initialized every time transaction restarts and function f called. Variable z denotes another class of local variables where the initial value is assigned outside of transaction and it is never changed inside the transaction. Variable "l" inside the function f can obviously be omitted for versioning. *In nutshell, the only local variables which will require versioning are the ones where the first memory request to them inside the transaction is a load and further it is also modified. This is the common scenario for shared variables but not for the locals.* Compiler optimizations can subsume variables into registers for performance. However, no compiler optimization technique changes the order of load and store to a memory location. Therefore this second-order dependency between load and store to a thread-local variable can be used reliably.

Also consider the fact that x, i, z and u are all mapped into the same cache line. We will present the complete algorithm in section V, which satisfies all these constraints and does a) Separation of true-shared and thread local data and b) Separation of thread local data which do not require versioning from a few local variables which do require versioning. Note that a local variable may be modified in several places inside different if-else sections with shared variables inside the conditions. Version can still be omitted for that. However, if it is modified-first in some sections and read-first in others where shared variables are in the condition, then that will lead to an unpredictable behavior in the program. Since programmer does not have a-priori knowledge about the abort/commit behavior of a program, sometimes the value of that particular local variable used inside a section will be the old value due to an abort. In other times it will be the committed value. Because of this reason such construct is improper and not found in any of the applications.

In our benchmark STAMP[3], as shown in Figure 6, we have found that on average only around 1% of the local variables fall under the category where old values for them must be maintained. For compilation, level 3 optimization was enabled in gcc. In the figure, we have counted the total number of local variables, including both local stack and local heap variables, accessed inside all the transactions. We then classify the result in two groups. One which does not require versioning and other which does require versioning. Data are normalized with respect to the prior value. Note that, there are addresses which are counted in both the categories. This happens because some local variables change its behavior from "version not required" to "version required" between different atomic blocks or in the same atomic block in different iterations. However, our proposed algorithm is not affected by either of these cases.

From our results, it seems a 128 entry fully-associative buffer to store old values of these few local variables in word resolution will be enough for almost all cases. If it overflows for some corner case applications (like in the case of ssca2), we can include those local variables in the commit packet.

### *E. Power Implication in L1 and LUB*

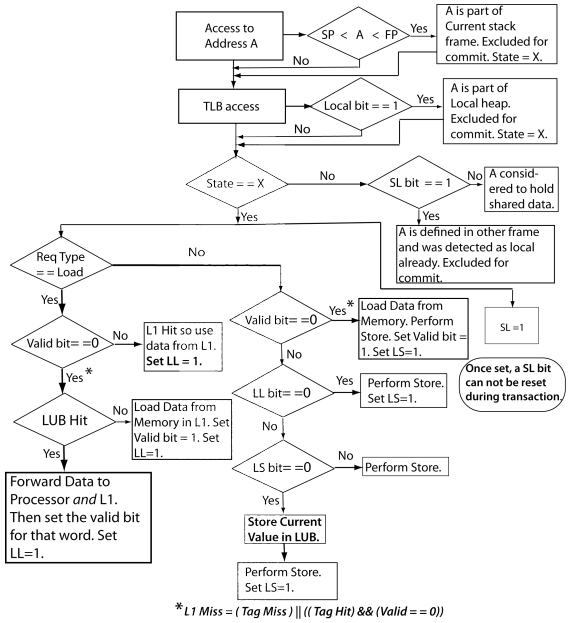
We used CACTI[12] for estimating the power increase in L1 because of adding these extra bits. We have proposed one R, W and SL bit for the entire line. Moreover we have proposed, Local Load (LL), Local Store (LS) and Valid bits per word in every cache line. For Alpha 21264, L1 data cache size is 64 KB, cache line size is 64 bytes and one word is 16 bits. So we need (32*3+3) = 99 extra bits per line.

Our CACTI simulation shows the power increment of L1 Data cache because of addition of 99 extra bits per line is *only* 6% in 65nm. The addition of 99 extra bits assumes local variables are padded to occupy one word. If every local variable is padded with cache-line granularity then we need only 3 bits per line.

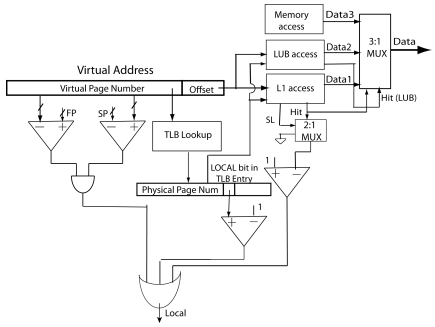
For CACTI simulation, LUB is assumed to be organized in line size of 8 bytes for a total of 256B with full associativity. Simulation result shows that read power of LUB is 60mW. On the other hand power consumption of the 64KB L1D read is 314mW. Therefore LUB consumes only 20% of the L1D power. For comparison, if we use a 32KB fully-associative cache for LUB to store old versions of all local variables, the power consumption is 6Watts. This is the reason we should reduce the LUB size and our technique helps here. Power consumed during a 2MB L2 read is 1512mW. Therefore, a small LUB for undo-logging is 25x times more power-efficient, apart from being 10x times faster. It is to be noted that because of our algorithm, the LUB size reduced 100x times making it amenable to practical implementation.

## V. The Proposed Algorithm

With these micro-architectural and OS modifications we now present the complete algorithm (Figure 7(a)). Whenever a processor performs a load or store within a transaction it checks the virtual address with the current stack boundaries. The lower bound of the stack is pointed by the Stack Pointer (SP) and the upper bound by the Frame Pointer (FP). If the virtual address is found to be pointing between these two registers then the address belongs to the current stack frame and is excluded for the commit. To store this information in the cache level we need to set the Speculative Local (SL) bit. However that can happen only after the TLB translation when the line is either found in the L1 as a L1 hit or brought




(a) The Flow-Chart Diagram




(b) Hardware Implementation

Fig. 7. Flow-Chart Diagram of the Algorithm and its Hardware Implementation

into L1 from the next level of memory. Therefore to keep the information that the virtual address has been detected as local till L1 can be accessed, we set a State variable to hold state $X$. After the TLB access is done, we have the information of the physical page and its associated protection mask. Now we have described in detail in section IV-B.1 on how to modify operating system kernel to propagate the local property of a physical page by setting one particular bit in the page protection bit field. After TLB translation, this bit can be checked. For our specific implementation involving Linux kernel and Alpha, the bit position chosen corresponds to the 21th bit in the Page Table entry. Therefore immediately after TLB translation if this bit is found to be set, then the address is guaranteed to belong to a local heap which was earlier allocated using our proposed allocator local_malloc(). So we disregard the address from commit set as well and set the State variable to hold state $X$ again. This State variable can be a simple 1 bit which will be accessed for every load and store inside a transaction.

If the State is $X$, then clearly the variable being accessed is local. In next phase we need to determine if this local variable needs versioning or not. To do that first the request type to it is checked. If it is a "Store" then we check the valid bit of it. If valid bit is unset then it denotes a L1 miss. So data are forwarded from L2 and Valid bit is set. Also for future reference that a local store took place, we set the LS bit. If valid bit is set, then the data are already present in L1. We check the LL bit to find out whether processor has used the data or not. If LL bit is set, then before doing the store, the data are inserted in LUB. This is required to keep the old version which will be required in case of an abort.

If the request type is Load, then again if Valid bit is set we use the data from L1. If valid bit is unset, that means either *it is a tag miss or the valid bit is unset because of an abort*. So we check LUB for the data. If it is found in LUB, it is forwarded from LUB to Processor and L1. In parallel SL bit for the line is set to 1. Once set, a SL bit can *not* be reset during transaction.

Next we go back to the scenario when after TLB translation, the state is *not* $X$. In that case, we further check the status of the SL bit. This is to handle all "pass by references" properly. If it is set, then even though the address is failing locality check at present, it was detected as local earlier since once a SL bit is set it can not be reset during a transaction. This implies that this variable was defined and accessed in some other frame and then it is also passed to the current frame by reference. So, we exclude it from the commit packet as well.

If it is unset, we consider that the address is pointing to a true shared data. The decision leaves the SL bit to be 0. However at a later point of time, it may happen that this address will pass the bound check with FP and SP and therefore will make a transition from the shared to local status. This scenario could arise if a function first immediately passes the variable as a reference to some other function and later modifies the return value in its own frame. In that case we will detect its locality when we reach that frame as the stack is getting wound.

At the end of the transaction all loads and stores to the local stack and local heap addresses will be identified and corresponding SL bits will be set. Variables like "p" in Figure 1(a) will be correctly treated as shared even if they belong to the program stack. This is because virtual address of p will not pass bound checks and also local bit will remain unset. A cache line gets included in the commit packet only if its corresponding SL bit *is not* set but the W bit *is* set which denotes that the transaction has modified that cache-line and it holds shared data.

In case of an abort, first all valid bits are cleared *except* if SL==1 and LL==1 and LS==0. This setting denotes the class of local variables (like "z" in Figure 5) which are only read inside the transaction but never modified. Therefore they do not need to be invalidated even in case of abort. Almost all thread-local pointers accessed inside transactions have this property. After this, all R, W, SL, LL and LS bits are cleared. In case of commit, first all transactional bits are flash cleared and all entries in the LUB are invalidated. However valid bits are not modified in L1 for local cache lines with SL bit set. On commit, valid bits of only cache lines which contain shared data (SL==0) are cleared, because in the next iteration new

updated values for them must be read from a non-speculative level of memory like L2. Since lines with SL bit set are excluded from the commit, this ends the commit phase for those lines. Now the next phase begins for the set of addresses which are truly shared among threads and therefore their values must be pushed at the L2 level of memory-hierarchy.

To illustrate the algorithm using an example, let us use the pseudo-code shown in Figure 5 in the previous section. Clearly, access to x, y, z, u, l and i will be detected as local. x, z, u, l and i will pass stack bound check whereas y will have the special bit set in the protection bit field in its TLB entry implying that it belongs to the local heap. However, the particular local variables such as i need to be treated differently. In the first iteration of the loop, x, z, u and i are all brought into L1 as one cache-line. Let us assume a scenario where the first iteration is a commit and second iteration is an abort. Therefore after commit valid bits are kept for all local lines, all LL and LS bits are cleared and the LUB invalidated. In the second pass, before the abort, as variable i is accessed for the first time, LL bit corresponding to it will be set. Next time when it is being incremented its old value (2) will be stored in the LUB and LS bit will be 1. Next load to i will come from the L1 because it will be a hit since the valid bit for it is already set. Next increment to i will not cause the value to be re-stored in the LUB because its LS bit is set now. After abort, when "i" will be loaded for the first time, it will be a L1 miss (because its valid bit is cleared) *but* it will be a LUB hit since we stored its value earlier.

To handle cases like x, after abort when x is stored for the first time, its valid bit will be set again. Therefore subsequent load to x will be satisfied from L1 itself. For variables like z, abort has no effect because its valid bit will not be cleared.

We have shown a partial hardware implementation of our proposed scheme in Figure 7(b). In this figure only the hardware components needed to separate shared and local data are shown. As can be seen, we only need four comparators, two of them are one bit, to generate an output named "Local". This output is pulled high for local addresses in the same cycle as L1 hit. This is possible because typically index and block offset access in L1 and TLB translation are done in parallel. Filtering of local variables not requiring versioning from local variables requiring versioning is not shown in the hardware implementation. The data to be forwarded to the processor are first checked in L1 or else come from LUB, if it is a L1 miss and LUB hit. Otherwise it is loaded from the memory. Note that LUB is *not* refilled like the miss-cache[8]. If data come from memory it is stored directly in the L1 not in LUB.

Also note how it can help if the transaction overflows L1. In case of "overflow", a processor can evict a line with SL bit set to 1. If that line contains some local variables which need versioning, then if the transaction subsequently aborts, in next iteration correct value will be found in the LUB. If the transaction commits, then in the next iteration, it will be a L1 miss (because of the eviction) and also be a LUB miss (because the earlier commit invalidated the LUB). However, still processor will get the correct value from the next level of memory. However, if the transaction re-touches an already *evicted* line while still in execution, it should bypass the LUB check and the line should be brought back to L1 (not shown in the figure).

Sometimes programmer may want to avoid using frame pointer for generating highly compact binary. In that case a separate register would be needed which should act like frame pointer. It needs to be managed by the processor whenever stack grows or shrinks.

## VI. Implementation Methodology and Results

### A. Simulation Environment and Applications

Simulations are done in a substantially modified version of the M5 full-system simulator[1] adding support for a lazy-lazy HTM, simulating Alpha 21264 architecture and running modified version of Linux kernel 2.6.13. Table I lists the important parameters used in the simulation. For the interconnect, we have used a Split-Transaction bus model to connect all cores.

TABLE I
Parameters used in simulation

| Feature | Description |
|---|---|
| CPU | 1-16 single issue in-order Alpha core frequency 1GHz |
| L1D | 64KB 64 byte line size 2-way assoc 1 cycle access |
| L2 | 2MB 64 byte line size 8-way assoc 10 cycle access |
| Interconnect | Common Split-Transaction Bus |
| Main Memory | 1GB, 100 cycle latency, Single Read/Write Port |
| Directory | Full-bit vector sharer list 10 cycle latency |

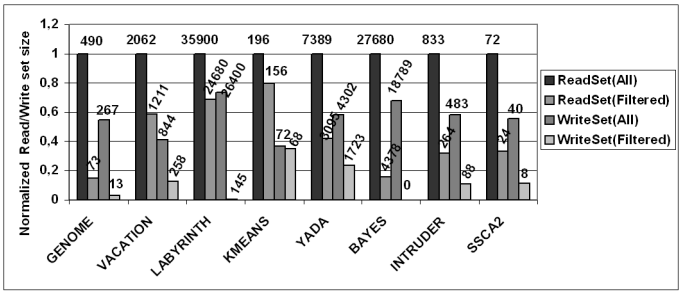



Fig. 8. Filtered vs Unfiltered Read/Write set size (in bytes)

For evaluation purpose we used applications from the STAMP benchmark suite[3]. Inputs used in the simulated runs are as suggested in [3].

### B. Results

Figure 8 shows the reduction we get in the transaction having the maximum readset and writeset size. Here we eliminate all local accesses regardless of whether they require versioning or not.

Figure 9 shows the speed-up obtained in the Scalable TCC[4]. The speed-up number with respect to the unfiltered

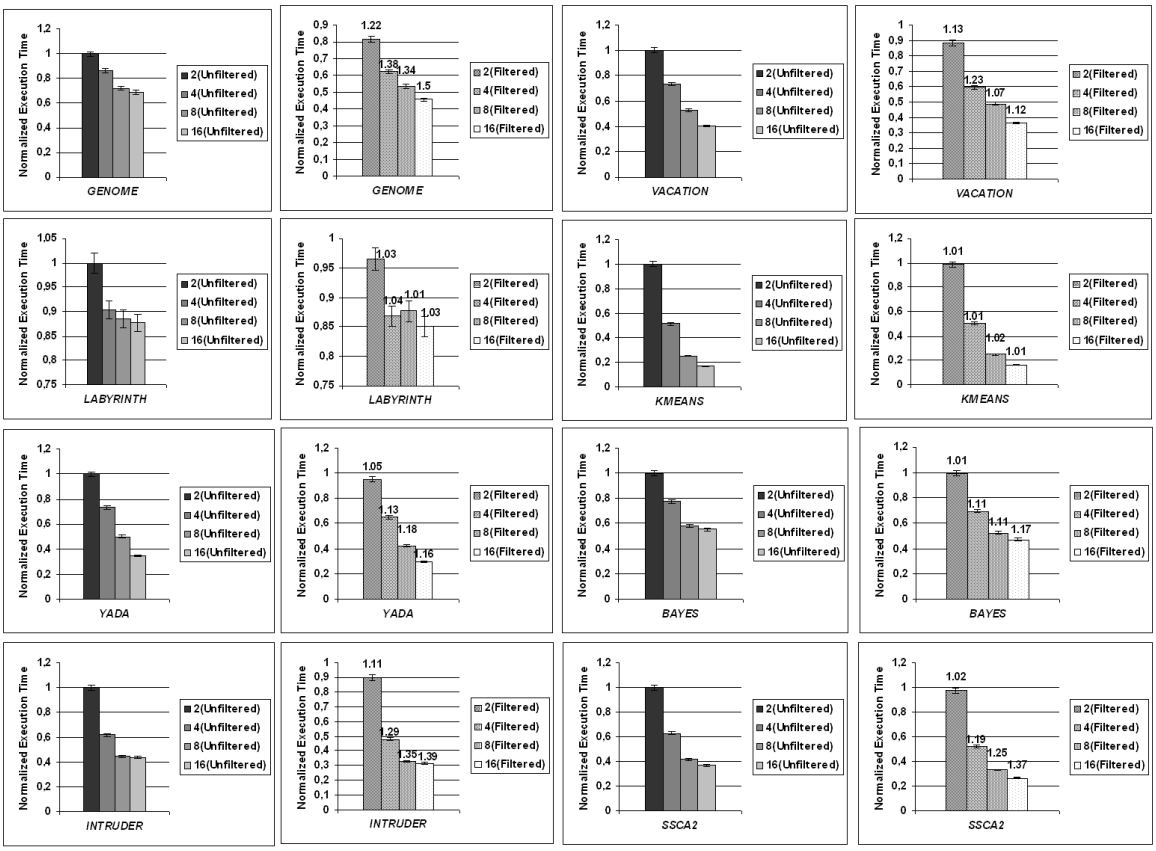



Fig. 9. Normalized Execution times showing speed-ups for Scalable-TCC

version is showed on top of the data representing the filtered version with the same number of processors configuration. A number *n* denotes a speed-up of *n*x times. All execution times are normalized with respect to the execution time for 2 processors configuration without employing any filtering. Figure 10 reports the speed-up obtained for the conventional version of the TCC. Across all applications and 2, 4, 8 and 16 thread configurations, the average speed-up obtained was 16% and 31% for the Scalable-TCC and the Bus-based Conventional-TCC respectively.

**B**ayes: This application highlighted a particularly interesting scenario. In this application we have a transaction which without any filtering becomes the transaction with the biggest writeset. However after filtering, its size reduces *exactly* to zero. This is because the transaction (*TMfindBestInsertTask* in *learner.c*) only scans a shared data structure (a shared queue) to find a suitable entry. Therefore every write it performs is modifying a local variable. With filtering we have reduced its writeset size from around 20000 bytes to 0.

## VII. Conclusion

In this paper we have outlined an original algorithm and implementation to filter out local accesses from an executing transaction. We have also presented in the same algorithm, how to dynamically identify local accesses which do not require versioning. Our algorithm integrates these two types of segregation (shared vs local and locals requiring versioning vs locals not requiring versioning) seamlessly into one. Our result shows that it is possible to elude versioning for 99% of local variables without compromising correctness. This leads to a power savings of 100x times on a hardware buffer proposed to store the undo-log while giving the same speed-up. This will be useful for the other eager-eager type of HTM[11] too. For such HTMs, several trips to the undo-log kept in a separate virtual memory location can be avoided.

Our HTM framework models the state-of-the-art in the lazy-lazy type of HTM, named Scalable-TCC[4]. Our proposal improves the main performance bottleneck of such HTMs which is the commit phase. We evaluate the proposal using state-of-the-art benchmark available for transactional memory named, STAMP[3]. With our proposal we got an average speed-up of 16% for the Scalable TCC and an average speed-up of 31% for the Conventional bus-based version of the TCC.

## Acknowledgments

This work is supported by the Barcelona Supercomputing Center (Centro Nacional de Supercomputación) and Microsoft Research Lab, Cambridge vide the collaboration contract no TIN2007-60625; by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC) and by the European Commission FP7 project VELOX (216852). Sutirtha Sanyal is also supported by a scholarship from the Government of Catalunya.

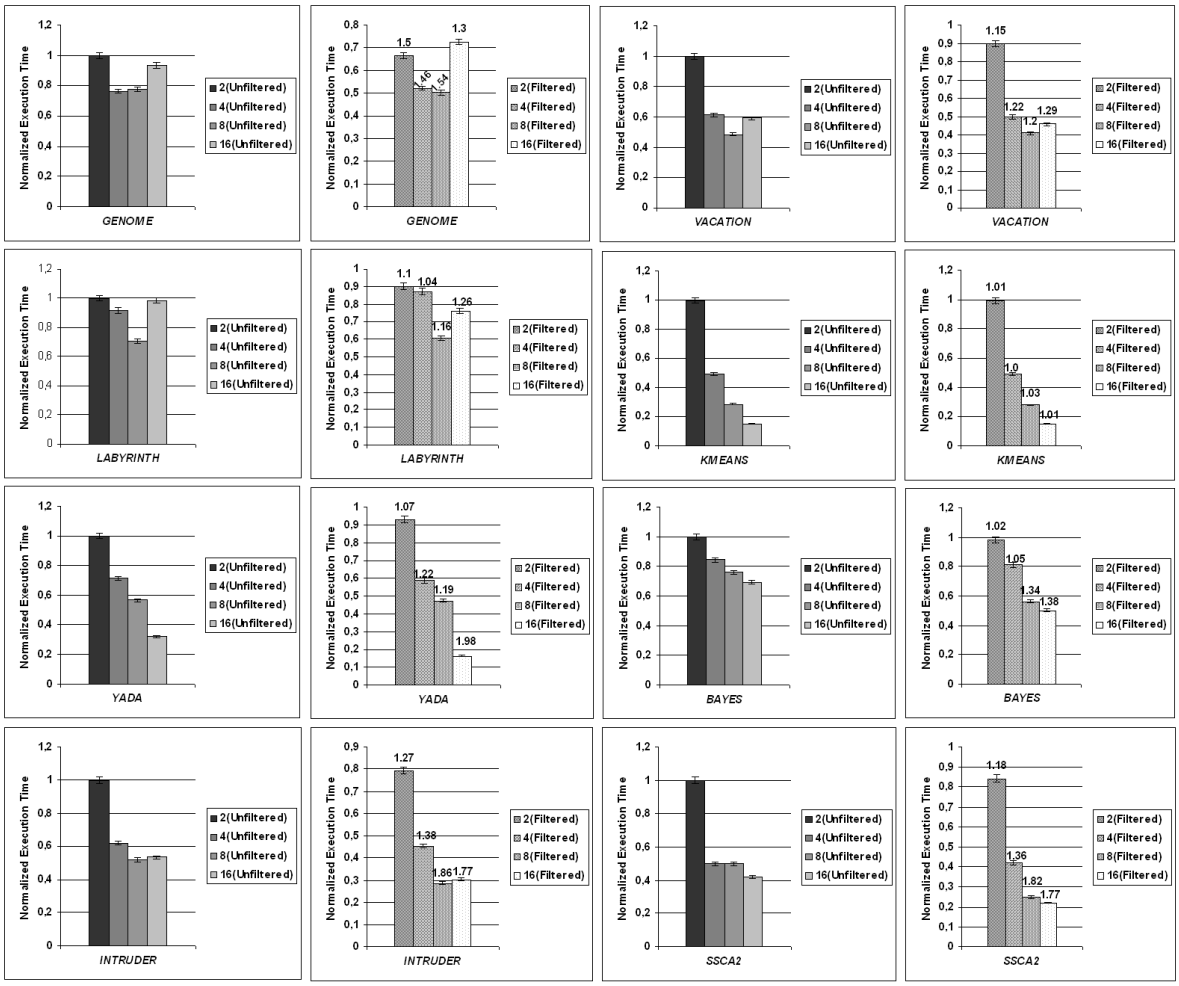



Fig. 10. Normalized Execution times showing speed-ups for Conventional-TCC

## REFERENCES


[1] N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S.K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE MICRO*, pages 52–60, 2006.

[2] C. Blundell, J. Devietti, E.C. Lewis, and M.M.K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. *Proceedings of the 34th annual international conference on Computer architecture*, pages 24–34, 2007.

[3] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[4] H. Chafi, J. Casper, B.D. Carlstrom, A. McDonald, C.C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A Scalable, Non-blocking Approach to Transactional Memory. *Proc. of the 13th Intl. Symp. on High Performance Computer Architecture, Phoenix, AZ, Feb*, 2007.

[5] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. *Proceedings of the 2006 ASPLOS Conference*, 41(11):336–346, 2006.

[6] L. Hammond, V. Wong, M. Chen, B.D. Carlstrom, J.D. Davis, B. Hertzberg, M.K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)*, 1063(6897/04):20–00.

[7] M. Herlihy and J.E.B. Moss. *Transactional memory: architectural support for lock-free data structures*. ACM New York, NY, USA, 1993.

[8] NP Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Computer Architecture, 1990. Proceedings. 17th Annual International Symposium on*, pages 364–373, 1990.

[9] S. Kumar, M. Chu, C.J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 209–220. ACM New York, NY, USA, 2006.

[10] D. Lea and W. Gloger. A memory allocator, 1996.

[11] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. LogTM: Log-based transactional memory. *Proc. 12th Annual International Symposium on High Performance Computer Architecture*, 2006.

[12] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman Jouppi. Cacti 5.3, 2008.

[13] L. Yen, S. C. Draper, and M.D. Hill. Notary: Hardware Techniques to Enhance Signatures. *Proceedings of the 41st annual International Symposium on Microarchitecture (MICRO)*, 2008.

[14] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. *TRANSACT: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.