

Architectural Support for Synchronization-Free Deterministic Parallel Programming

Cedomir Segulja and Tarek S. Abdelrahman
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
{seguljac, tsa}@eecg.utoronto.ca

Abstract

We propose a novel synchronization mechanism called versioning. It dynamically establishes a deterministic order of memory accesses in parallel programs that have serial semantics, in a way that is transparent to the programmer. This order is created in a distributed manner and is enforced by monitoring memory accesses and stalling threads if necessary. Versioning gives rise to parallel programming models in which programmers need not explicitly synchronize threads and only need to specify shared data, which greatly simplifies parallel programming. However, versioning introduces overheads and thus demands architectural support. We describe versioning and the architectural support it needs. We also propose one parallel programming model that utilizes versioning and use it to parallelize 13 benchmark applications. We build an FPGA prototype of a multiprocessor system with versioning support and show that good parallel speedups are obtained. Our analysis shows minimal impact of versioning, both in terms of timing overheads and in terms of additional hardware.

1. Introduction

Chip multiprocessors remain difficult to program and easing this task is currently a major challenge. Not surprisingly, there have been several software- and hardware-based approaches trying to tackle this challenge. Examples include parallel programming libraries, such as Intel Threading Building Blocks (TBB) [19]; compiler directive models, such as OpenMP [7]; language extensions, such as Cilk [14] and Deterministic Parallel Java (DPJ) [5]; and hardware-supported transactional memory proposals [15, 27].

One promising approach to program parallelization is the use of language extensions, such as those of Cilk and Cilk++ [16], to create parallel programs with *serial semantics*. That is, parallel programs that can be understood and executed as serial programs [17]. The parallel execution of such programs starts with a single thread, which executes the main function. During execution, a function call dynamically spawns a child thread that executes the function asynchronously with its caller. While parallel programs with serial semantics cannot express certain types of parallelism

(e.g., producer-consumer), their paradigm allows incremental program parallelization and hence eases the development of parallel programs. Unfortunately, existing language extensions still require programmers to synchronize accesses to shared data manually, which is hard and error-prone. They also result in parallel programs that are non-deterministic, which complicates parallel program debugging and deployment [10, 9, 18].

In this work, we present a novel synchronization mechanism called *versioning*. This mechanism provides deterministic execution for parallel programs with serial semantics and alleviates the need for programmers to synchronize threads. In versioning, each shared memory location is assigned a version number. Each parallel thread also receives its own version number for each shared memory location it accesses. During parallel execution, memory accesses are monitored. A thread is allowed to access a shared location if and only if its own version number for that location matches the current version number of the location; otherwise, the thread waits. Version numbers are created and updated in a distributed fashion and transparently to the software. They are created in such a way that the order of accesses in parallel execution is equivalent to the order of accesses in serial execution.

Versioning gives rise to a non-speculative parallel programming model in which there is no explicit synchronization of threads and thus no need to deal with atomicity or mutual exclusion. It only requires programmers or compilers to indicate shared data accessed by the functions targeted for parallel execution; dependences are automatically and dynamically detected and enforced. This greatly simplifies the tasks of parallel programming. Further, versioning results in deterministic parallel execution in which any parallel execution of a program produces the same output for the same input as serial execution.

Versioning does introduce overheads that stem from monitoring of memory accesses, from the comparisons of version numbers, and from the need to store version numbers. Thus, we introduce architectural support that alleviates these overheads. It consists of dedicated on-chip storage and a small logic, designed as a coprocessor. The proposed support is modular, and requires no changes to the processor pipeline, to the caches, nor to the coherence protocol. Indeed, in a prototype system, versioning structures require the equivalent of

about 15% of the 16KB L1 data cache per processor.

In the paper, we describe our versioning synchronization mechanism (Section 2) and its architectural support (Section 3). We propose one possible task-level programming model that is similar to TBB or Cilk and that utilizes versioning to eliminate the need for programmers to synchronize threads but requires them to specify shared data (Section 4). We build an FPGA prototype of an 8-processor system with support for versioning and evaluate the performance of 13 benchmark applications parallelized using this model (Section 5). The geometric mean of the speedups on 8 processors is 4.09x for all benchmarks, and 5.95x for 8 applications that are not memory-bound. We discuss related work in Section 6, and then present concluding remarks (Section 7).

2. Versioning

We first describe the basic mechanism, then extend it to support concurrent reads and to dynamically adjust the granularity at which dependences are monitored. The main data structures and the algorithms of versioning are then described.

2.1. Basic scheme

In versioning, every shared memory location is assigned a *version number*. Shared memory locations are identified *during run-time*, using compiler- or programmer-provided information about data accesses (for example as described in Section 4). The version number of a location is created and initialized to 0. As execution progresses, the version number of a location is gradually increased, finally reaching a value W , where W is a large integer.

Each thread maintains a pair of numbers (an, rn) for each shared location it accesses: an *acquire number* and a *release number*. During execution, memory accesses are monitored; a thread is allowed to access a shared location if and only if its acquire number for that location matches the current version number of the location; otherwise, the thread waits. Upon finishing, a thread updates the version numbers of all its shared locations using its corresponding release numbers.

The acquire and release numbers of a thread are created when the thread is created and are assigned values in such a way that threads that access a certain location *earlier* in sequential execution receive smaller acquire numbers for that location than threads that access the location *later* in sequential execution. Furthermore, the release number of a thread for a location is always larger than its own acquire number for the same location and it matches the acquire number of the thread that accesses the location immediately later in sequential execution. This ensures that threads access each location in sequential order and that deadlock cannot occur. The algorithm for assigning these numbers is described in Section 2.4.

Versioning is illustrated using the example shown in Figure 1a. The `main` function writes to the global variable `x`, then invokes the function `childA`, and eventually returns the value of `x`. `childA` increases the value of `x`, invokes the function `childB`, reads `x`, and then returns. `childB`

only reads `x` and returns. Since `x` is read and written by the three functions, any parallel execution must preserve sequential execution order with respect to the variable `x`.

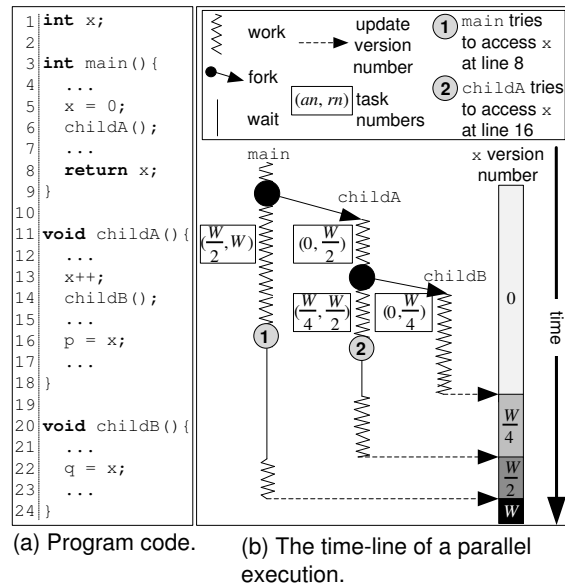


Figure 1. An illustration of versioning.

The parallel execution of the above example is depicted in Figure 1b. The first access to `x` in `main` proceeds unchanged. When `childA` is invoked, a new thread is created to execute it. Since `childA` will access a shared variable `x`, a version number used to synchronize accesses to `x` is created and initialized to 0. Also, the new thread receives the numbers $(0, \frac{W}{2})$ while `main` receives the numbers $(\frac{W}{2}, W)$ for `x`. This signals that `childA` must access `x` before `main` does. Thus, when `main` attempts to access `x` again while `childA` is still executing, as shown in the figure, it must wait since its acquire number does not match the version number of `x`.

In contrast, `childA` is able to access `x`. It then creates a thread that executes `childB`. The acquire and release numbers of both `childA` and `childB` are updated as shown in Figure 1b. Thus, when `childA` attempts its second access to `x` while `childB` is still running, it must wait because of the mismatch between its acquire number and the version number of `x`. However, when `childB` completes execution, it updates the version number of `x` to its release number, or $\frac{W}{4}$. This matches the acquire number of `childA` at this point, allowing it to continue and to read `x`. When `childA` in turn completes execution, it updates the version number of `x` to its release number, $\frac{W}{2}$, which allows `main` to continue. Thus, the three concurrently executing threads, `main`, `childA` and `childB` access the variable `x` according to sequential execution order.

2.2. Supporting concurrent reads

The order of reads between two sequentially consecutive writes can be freely interchanged, allowing a greater degree of parallelism. In the above example, `childA` need not wait when it attempts to read `x` at line 16; rather, it should be

allowed to proceed concurrently with `childB`. The basic versioning scheme can be easily extended to support concurrent readers. The version number for each shared location is made to consist of two components: a *write version* (w) and a *read version* (r). Further, in addition to the acquire and release numbers, each thread maintains a *delta number* (dn) for each shared location that it accesses. These additional components are used to serialize consecutive writes but allow multiple readers in-between to execute concurrently.

A thread can write to a memory location only if (1) its acquire number for that location matches the current write version of the location and (2) if its delta number for that location, when *added* to the read version of the location, gives a value R , where R is a large integer. In contrast, for a thread to read a memory location, it suffices that its acquire number matches the current write version of the location. When a thread finishes, it sets the write versions of all the shared locations it wrote to its release numbers and sets the read versions of these locations to 0. However, for the locations that the thread only read, it increases their read versions by the delta numbers that it maintains for those locations.

The acquire number that a thread maintains for a location it reads or writes is made equal to the release number of the thread that, in sequential execution, *writes* to that location earlier. Consequently, during a parallel execution, a read or a write can be executed only after the sequentially earlier write has finished. Further, all threads that in sequential execution consecutively read a location receive the same acquire number for that location. This number is also equal to the release number of the preceding thread that writes to that location. This allows readers to proceed concurrently. Finally, the delta number that a thread maintains for a certain location is assigned in such a way that only when all readers add their delta numbers to the read version of the read location, the result equals R less the delta number of the subsequent writing thread in sequential execution. This ensures that a write occurs only after all sequentially earlier reads finish. Section 2.4 gives the algorithm for assigning the acquire, release and delta numbers.

Consider again the example shown previously in Figure 1a. The parallel execution when concurrent reads are supported is depicted in Figure 2. The acquire, release, and delta numbers that a thread maintains are denoted with a triplet (an, rn, dn) . When the thread executing `childA` is created, it receives the numbers $(0, \frac{W}{2}, R)$, while the numbers of `main` are set to $(\frac{W}{2}, W, R)$. When `childA` first accesses x , both conditions needed to execute a write hold, and the access is executed. When `childA` creates a thread that executes `childB`, the acquire number of `childA` is unchanged, while the delta number is split between the two threads, since `childB` only reads x . Thus, `childA` will be allowed to read but not write x . Once that `childB` completes, it increases the read version of x by its delta number, $\frac{R}{2}$, allowing `childA` to also write to x .

2.3. Structures

Versioning uses three types of structures: a *global version table* (GVT), a *local version table* (LVT), and an *access table*

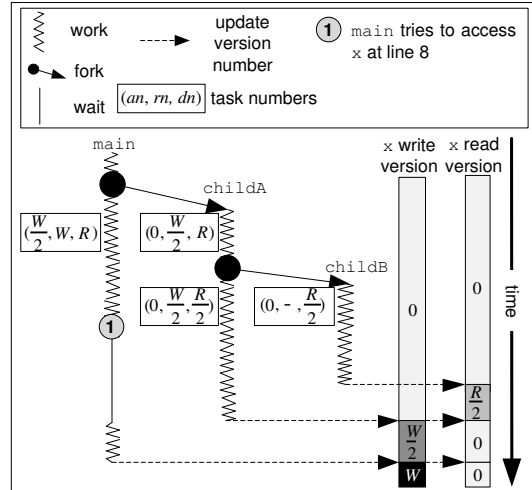


Figure 2. A parallel execution of the example from Figure 1a when concurrent reads are supported.

(AT). The GVT is shared by all threads and it contains an entry for each version number in the system. Additionally, a global counter, called the *version counter* (vc), maintains the total number of version numbers. Initially, the GVT is empty and vc is 0. When a new version number is added to the GVT, its write and read components are set to 0, and vc is incremented.

An LVT contains versioning information private to a thread. An LVT entry represents a memory location accessed by a thread and it consists of 7 fields ($addr, re, i, an, rn, dn, a$), where $addr$ uniquely identifies an entry and denotes the location's address, re is a *read* bit indicating that the location is only read by the thread, i is index to the GVT used to access the version number of the location, an, rn, dn , are the previously explained acquire, release and delta numbers, respectively, and a is an *acquired* bit that designates if the thread can access the location without comparing version numbers, as it will be explained below.

The LVT of a thread is initialized when the thread is created using the AT of that thread. An AT entry represents a memory location that will be accessed by the thread, and it consists of 2 fields ($addr, re$), which have the same meaning as the corresponding fields of an LVT entry. The AT is created from the data access summaries given in the sequential code and the procedure of creating it is described in Section 4.

2.4. Algorithms

Versioning uses three algorithms: the *acquire*, the *release* and the *generate algorithm*, the combination of which ensures that the sequential execution semantics are preserved.

The acquire algorithm is used to check if a thread t is allowed to access a memory address $addr$. Its pseudo-code is shown in Figure 3. The *read* parameter has a value of 1 in a case of a read access, and 0 otherwise. The acquired bit is used to reduce the number of comparisons of version numbers. Once a thread successfully accesses, i.e., *acquires*, a

memory location, the acquired bit is set to 1. On subsequent accesses to the same location, no version checks are necessary. Since a set acquired bit allows both for reads and writes to proceed, care must be taken when a thread reads from a location at a certain execution point, but it may write to it later on. At that execution point, the acquired bit cannot be set, since a stronger condition needs to hold for a write. Hence, an additional check is made at line 5.

```

Acquire( $t, addr, read$ )
1: if ( $\exists l \in LVT_t$  such that  $l.addr == addr$ ) then
2:   if ( $l.a == 1$ ) then return; # Already acquired?
3:   if ( $read == 1$ ) then
4:     while ( $l.an \neq GVT[l.i].w$ ) do wait;
5:     if ( $l.re == 1$ ) then  $l.a = 1$ ; # Set acquired only if read-only
6:   else
7:     while ( $l.an \neq GVT[l.i].w$ ) || ( $l.dn + GVT[l.i].r \neq R$ ) do wait;
8:      $l.a = 1$ ;

```

Figure 3. The acquire algorithm.

Upon completion, each thread updates the version numbers of all the memory locations that exist in its LVT. This action is done by the release algorithm, which is shown in Figure 4. The process of updating version numbers has been previously discussed, but there are two subtle points. First, before a thread can update the version number of a location that it reads/writes, it must ensure that all the threads that write/access that location earlier in sequential execution have already finished. This is enforced by invoking the acquire algorithm (line 2). The invocation of the acquire algorithm at this point can never lead to deadlock since earlier threads in sequential execution are always assigned smaller or equal acquire numbers. For example in Figure 2, `childA` must check that `childB` has finished before updating the version number of x . Second, the addition at line 4 must be done atomically, since readers can concurrently try to increase the same read version.

```

Release( $t$ )
1: for all ( $l \in LVT_t$ ) do
2:   Acquire( $t, l.addr, l.re$ ) # Wait for "earlier" threads
3:   if ( $l.re == 1$ ) then
4:     atomic { $GVT[l.i].r = GVT[l.i].r + l.dn$ ;}
5:   else
6:      $GVT[l.i].w = l.rn$ ;
7:      $GVT[l.i].r = 0$ ;

```

Figure 4. The release algorithm.

The generate algorithm defines how acquire, release and delta numbers are created, and its pseudo-code is shown in Figure 5. The algorithm is run at thread creation points: it initializes the LVT of the child thread (t_c) and updates the LVT of the parent (t_p), using the AT of the child as input.

If a memory location from the AT does not exist in the LVT of the parent thread, no sequentially earlier threads access that location. Hence, a new version number is created to synchronize accesses of the parent and child threads to the location and a new entry is added to the LVT of the parent thread (lines 5-8). Since the version counter is shared by all threads, the actions at line 5 must be done atomically.

For locations that already exist in the LVT of the parent thread, or that have just been inserted, the algorithm modi-

```

Generate( $t_p, t_c$ )
1: for all ( $e \in AT_{t_c}$ ) do
2:   if ( $\exists l' \in LVT_{t_p}$  such that  $l'.addr == e.addr$ ) then
3:      $l = l'$ ; # An entry already exists for this location
4:   else
5:     atomic { $i = vc$ ;  $vc = vc + 1$ ; } # Create a new version number
6:      $GVT[i].w = GVT[i].r = 0$ ;
7:      $LVT_{t_p}[e.addr] = (e.addr, 0, i, 0, W, R, 1)$ ;
8:      $l = LVT_{t_p}[e.addr]$ ; # Created a new entry for this location
9:   if  $e.re == 0$  then
10:     $LVT_{t_c}[e.addr] = (e.addr, 0, l.i, l.an, \lfloor \frac{l.an+l.rn}{2} \rfloor, l.dn, l.a)$ ;
11:     $LVT_{t_p}[e.addr] = (e.addr, l.re, l.i, \lfloor \frac{l.an+l.rn}{2} \rfloor, l.rn, R, 0)$ ;
12:   else
13:     $LVT_{t_c}[e.addr] = (e.addr, 1, l.i, l.an, -, \lfloor \frac{l.dn}{2} \rfloor, l.a)$ ;
14:     $LVT_{t_p}[e.addr] = (e.addr, l.re, l.i, l.an, l.rn, \lceil \frac{l.dn}{2} \rceil, l.a \wedge l.re)$ ;

```

Figure 5. The generate algorithm.

fies the entry of the parent LVT and adds a new one to the LVT of the child. The floor and ceiling functions are used to ensure that all numbers used in versioning are integers. If a location will be written to by the child, the release number of the child and the acquire number of the parent are set to $\lfloor \frac{l.an+l.rn}{2} \rfloor$ (lines 10-11). This ensures that the release number of the child (and the updated acquire number of the parent) is different than all previously generated acquire and release numbers for the version number begin considered. For read-only locations, the algorithm ensures that the sum of the delta number of the parent and child remains the same as the original delta number of the parent (lines 13-14).

Two things require clarification. First, at lines 10-11, it must hold that $l.rn - l.an > 1$ in order to generate a unique number (similarly it must hold that $l.dn > 1$ at lines 13-14). Hence, a reasonable large W and R need to be chosen. If the condition does not hold, a child thread cannot be created, since sequential execution semantics can no longer be guaranteed. Second, any integer from interval $(l.an, l.rn)$ can be chosen instead of $\lfloor \frac{l.an+l.rn}{2} \rfloor$. Assuming that both the parent and the child thread will subsequently create an equal number of threads that will access the location being considered, choosing the middle value maximizes the number of threads that can be created. However, if it is known that the child thread will not create any threads, $l.an + 1$ can be used instead of $\lfloor \frac{l.an+l.rn}{2} \rfloor$.

2.5. Variable-granularity versioning

It is not practical to assign a version number to every single memory location accessed by threads. Instead, a version number is assigned to a *region* of memory locations. Thus, sequential execution semantics are enforced on the basis of these regions rather than on the basis of individual locations. The size of these regions is referred to as *granularity* at which versioning operates. Choosing a coarser granularity results in a smaller number of version numbers, and consequently, reduces the space needed to hold the GVT, LVTs, and ATs. However, a coarser granularity may limit parallelism due to false sharing. A scheme that uses the same granularity of all regions throughout the entire execution is referred to as *fixed-granularity* versioning, and it operates as described in the previous section. It is also possible to have *variable-granularity* versioning in which the granularity is

inferred from functions' data accesses. For example, if the user specifies that a function accesses a certain section of an array, a single version number will be used to enforce the proper ordering of accesses to the entire section. If however, during the execution of this function a child thread that accesses only a subset of the array's section is spawned, the granularity is dynamically adjusted to allow the parent thread to concurrently access the subsection of the array not being accessed by the child.

This scheme is illustrated using the example shown in Figure 6. For a single LVT entry, a thread can compare the acquire number to one version number, and use the release number to update a different version number. Hence, the acquire and release numbers are written in a form (an_{i_A}, rn_{i_R}) to denote that the acquire number an is compared against the version number i_A and that the release rn is used to update the version number i_R . Also, for some LVT entries, a thread may not update any version number; the symbol '-' is used for i_R in that case. When main creates a thread that executes `childA`, the version number 1 is created, which is (at this phase) used to synchronize accesses to the entire array `a`. Additionally, an entry is added to the LVTs of the threads executing `main` and `childA`, as described earlier. However, since `childB` accesses only the last 10 elements of the array `a`, once the thread that executes `childB` is created, a new version number is also created. Version number 2 will be used to ensure that `childA` accesses the last 10 elements of the array `a` only after `childB` finishes. Further, a new entry is inserted into the LVT of `childB` while the previous entry in the LVT of `childA` is fragmented into two entries that correspond to the first 90 and the last 10 elements of the array. By doing so, `childA` is allowed to increase the first 90 elements of the array in parallel with the execution of `childB`. However, when `childA` attempts to access `a[90]` it must wait because of the mismatch between its acquire number and version number 2.

In effect, variable-granularity versioning resulted in identical execution to when fine granularity (a version number for each array element) is used, but with lower space requirements. The variable-granularity scheme uses the same types of structures as fixed-granularity versioning. No changes to the GVT are required, while new fields must be added to the LVT entries. The acquire and release algorithms are almost identical to the fixed-granularity case, while the generate algorithm is more complex, having to deal with the corner case when an LVT entry that uses different version numbers during the execution of acquire and release algorithms needs to be fragmented. Even then, versioning retains its distributed nature requiring that only the LVT of the parent thread and the LVT of a newly created thread are modified. Due to space considerations, we omit the algorithmic details.

2.6. Overheads of versioning

The implementation of versioning in a system introduces both space and time overheads. It requires additional memory space to store the GVT, as well as the LVTs and the ATs for each thread. The sizes of these structures increase with the size of shared data and the number of threads, and are

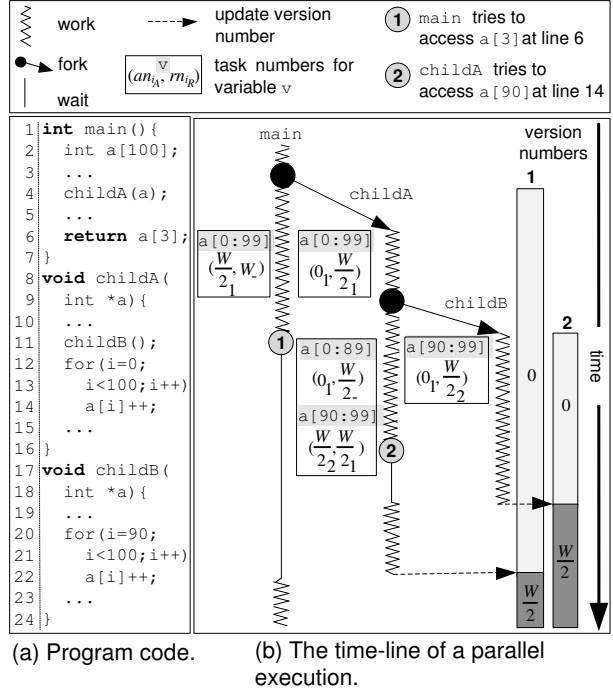


Figure 6. Illustration of variable-granularity versioning.

also determined by the versioning granularity. We expect the space requirements to be reasonable for most applications.

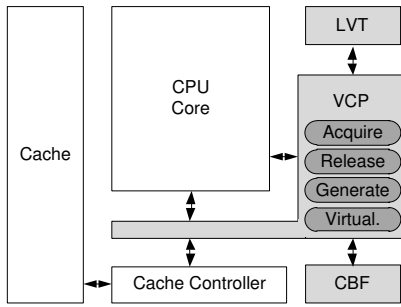
Versioning introduces time overheads because of thread management and the additional delay incurred during the execution of memory accesses. These overheads are summarized in Table 1. Upon the creation of a thread, the AT must be populated. Further, as will be described in the next section, a thread context switch will require the storing and loading of the thread's LVT. Most seriously, each memory access requires an LVT access and a GVT access. Should this overhead be incurred on every memory access, it would be detrimental to performance. Therefore, we introduce architectural support to mitigate this overhead, which is described next.

Table 1. Versioning overheads.

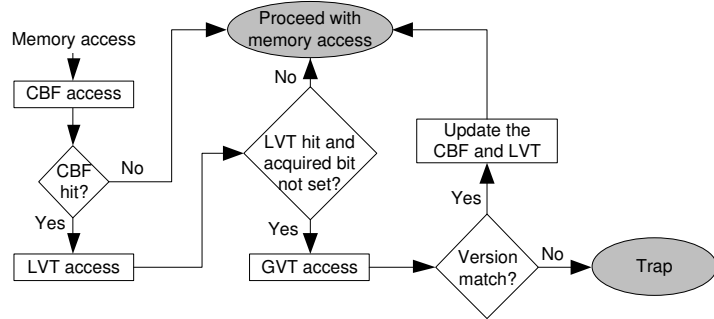
	Overhead	When does it occur?
Thread management	Populate the AT	Thread creation
	Exec. generate algorithm	Thread creation
	Load the LVT	Thread switch
	Store the LVT	Thread switch
	Exec. release algorithm	Thread end
Executing memory accesses	LVT and GVT lookups	Load/store inst.

3. Architectural support

Figure 7a depicts the placement of the main components of the hardware support for versioning in a typical micro-processor architecture. Our design requires two dedicated on-chip memories: one memory block holds the LVT of the currently running thread, while the other one is used as storage for a *Counting Bloom Filter (CBF)* [12]. A CBF is a fast, space-efficient structure that is used to conservatively



(a) The main components of the proposed support in a typical microprocessor architecture.



(b) The execution of a memory access in the presence of the proposed support.

Figure 7. Architectural support for versioning.

test membership of a set. A CBF may indicate that an element is a member of the set, when it is not (*false positive*). However, it never indicates that an element is not a member of the set when it is. In our implementation, the CBF holds the set of memory locations not yet acquired by the running thread and it is used to reduce the number of accesses to the LVT, as described below. No on-chip resources are dedicated to the GVT nor to the version counter; these are shared structures and are placed in the shared memory.

The *Versioning Coprocessor (VCP)* interfaces the CPU core in two ways: as a coprocessor that executes instructions added to the instruction set architecture (ISA) that implement versioning algorithms, and as a proxy to the cache controller that monitors memory accesses. In order to access the GVT, the VCP contains a register (not shown in the figure) that holds the address of the GVT.

3.1. The four main mechanisms of the VCP

The VCP coordinates the execution of memory accesses, transfers the LVT to/from the memory, updates the GVT, and executes the generate algorithm. These mechanisms are described next.

The VCP coordinates the execution of a memory access, as depicted in Figure 7b. The CBF is accessed on each memory access, and in parallel with cache lookup. A CBF miss implies that the memory access can be executed without accessing the LVT and GVT. Since the CBF holds only those memory locations not yet acquired by the running thread, the CBF mitigates the overhead of accessing the LVT and the GVT for each memory access. In the case of a CBF hit, the LVT is searched for a corresponding entry. Due to the CBF false positives, the requested entry may not exist or its acquired bit could already be set, in which case the memory access can safely proceed. Otherwise, the GVT address register and the acquire index of the LVT entry are used to calculate the address of the corresponding version number, which is then read from the GVT. In the case of a *version match*, the original memory access is resumed and the LVT and the CBF are updated. Otherwise, there is a *version mismatch*, a trap is raised and the program control is transferred to a trap handler. The handler invokes a scheduler to decide if the faulting instruction should be re-executed or the cur-

rent thread should be taken off the processor and replaced with a different one. With the use of CBF, the LVT and GVT lookups are made only when CBF hits occur. Consequently, the overhead of accessing memory accesses occurs only on CBF false positives, on version matches and on version mismatches. Our evaluation shows that CBF hits are infrequent and that the resulting overhead is minimal.

Next, the VCP contains simple logic that transfers the contents of the on-chip LVT to/from the memory. When a thread is taken off the processor, the VCP saves the contents of the on-chip LVT to memory, as part of the thread's context. The content of the CBF is not saved, since it can be derived from the saved LVT.

The VCP also executes the release algorithm. Before a thread finishes, the VCP walks through all the entries of the on-chip LVT and updates the GVT accordingly. The addresses of version numbers that need to be updated are derived by the VCP by adding the GVT base address register to the release index element of the LVT entry being considered.

Finally, the last mechanism of the VCP executes the generate algorithm. When a currently running thread creates a child thread, versioning requires that only the content of the LVT of the parent thread is modified, and that the new LVT is created for the child thread. Dedicated hardware reads entries of the AT of the child thread (from the memory), updates the LVT of the parent thread (on-chip), and saves the newly created entries of the LVT of the child thread (to the memory). In addition, the content of the CBF is made consistent with the LVT of the parent thread. Since the on-chip LVT has a limited size, it may fill. When the generate algorithm fails to insert a new entry, the child thread is not created and the corresponding function call is executed serially by the parent thread. This also occurs when a version number cannot be generated, as described in Section 2.4.

Extensions to the ISA are required in order to expose the hardware support to software. Newly added instructions trigger loading/storing the LVT, and the execution of the release and the generate algorithms.

In the proposed design, the CBF is accessed in parallel with the processor's cache. In order to hide these accesses from the processor pipeline, the result of the CBF access must be known no later than the cache tag comparison is done. Since the CBF is a much smaller structure than the

cache, it is sufficiently fast and the processor pipeline is oblivious of CBF accesses. Memory accesses that require the LVT and GVT lookups are presented to the processor core as cache misses, stalling the pipeline until the comparison of version numbers is done. A version mismatch is treated the same way as any other instruction-induced exception.

To read and modify the contents of the GVT, and transfer the on-chip LVT to/from memory, the VCP must be able to execute memory accesses. This is done by multiplexing the input signals of the cache controller; the cache input signals are driven by the processor core when the VCP is not executing one of the versioning instructions nor checking a memory access; otherwise, the signals are driven by the VCP.

4. A programming model

Versioning induces a deterministic parallel programming model in which programmers do not explicitly synchronize threads. As such, it can be used to improve the programmability of existing programming models that have serial semantics. Since versioning alleviates the need for user-level synchronization but requires the specification of data accesses, we opt to use an annotation-based model that is similar to that of Cilk/Cilk++.

In our model, programmers start from a sequential C code, and use two simple pragmas: `parallel` and `access`. Programmers insert the `parallel` pragma before the function invocations that may execute asynchronously in parallel. They also identify *side-effects* of these invocations. A function is said to have a side-effect if, in addition to producing a value, it has an observable interaction with the rest of the code. This interaction takes place through accessing global variables, local variables of the caller function(s), or dynamically allocated memory. Programmers describe these accesses by placing the `access` pragma before the function *definition* or *declaration*, followed by a C-like expression identifying a variable or an array section, expressed in terms of parameters and global variables (e.g., `#pragma access x` or `#pragma access a[0:9]`). Compiler support could be used to produce these pragmas, but such support is outside the scope of this work.

In addition to describing simple accesses, a compact notation is used to represent accesses to recursive data (e.g., linked lists or trees) found in many programs. For example, a traversal of a linked list pointed to by a pointer `head` is described with the expression `*(head->next*)`, summarizing accesses to `*head`, `*(head->next)`, `*(head->next->next)`, etc., until a NULL pointer is encountered. However, if it is known that a recursive structure is always accessed through the same *entry points*, then it is not necessary to describe accesses to all elements of the structure. For instance, for a function that modifies all the nodes of a tree, it suffices to only include the root node in its side-effects. In order to modify a tree node, the root node must be accessed first and proper ordering of accesses will be maintained.

Functions' side-effects are used at run-time to create ATs by mapping the variables reported by the `access` pragmas to memory locations that those variables occupy. To ensure

correctness, the side-effects of all callee functions must be incorporated in the side-effects of the caller. Further, all possible accesses must be conservatively specified, even if some of them may not take place during every invocation a function. Finally, deallocation of the dynamically allocated memory is also considered to be a side-effect, and it should be specified as a write to the memory location that will be deallocated. This is necessary to ensure that the memory can be freed only after all sequentially earlier threads have finished. Memory allocation, however, need not be reported, because the only legal way to use allocated memory is to pass its address through an existing variable.

Versioning operates at the memory address level (in hardware) rather than at the language level. Thus, it ensures that the order of accesses to a location is equivalent to their ordering in sequential execution, even if the location is accessed through different symbolic names in the program. Therefore, aliasing poses no problems to our scheme, which is one of its strengths.

5. Experimental evaluation

We evaluate versioning using a prototype implementation. We (i) explore the feasibility of the hardware support, (ii) quantify versioning overheads, (iii) investigate the effort of parallelizing applications using the proposed programming model, (iv) analyze the performance of applications under versioning, and (v) place our results in context by comparing the performance of the benchmarks using versioning to that using a representative software-only model, TBB, run on our system as well. TBB requires more parallelization effort, which is difficult to quantify. Nonetheless, the comparison allows us to show that versioning achieves comparable speedups to a software-only system. It would be ideal to compare the performance of our hardware-based approach to that of a software-only one that implements our programming model, and thus requires the same parallelization effort. Short of implementing versioning in software with excessive overheads, such a software-only approach does not exist.

5.1. Experimental setup

Our prototype, which we refer to as ROKO, consists of a precompiler, a run-time, and a versioning-enabled hardware, implemented in a Field Programmable Gate Array (FPGA).

The ROKO precompiler is a source-to-source compiler, implemented using Clang [8]. It takes a C program annotated with `parallel` and `access` pragmas and produces C code that can be executed on a multiprocessor with versioning support. The precompiler identifies the function calls marked with `parallel` pragmas and parses the expressions following the `access` pragma. The annotated function calls are replaced with the code that handles thread creation. Parallel functions are also rewritten to have a standardized interface, so that they can be handled by the run-time. If a function returns a value, it is transformed to a semantically equivalent one that does not, by adding a parameter to the parameter list and by writing the return value to the memory pointed by this parameter immediately before the function's

exit points. The newly added write is automatically added to the function’s side-effects.

The run-time is a thin software layer that provides an interface between the applications and the shared memory multiprocessor (SMP) with hardware support for versioning. It controls the creation, scheduling, and termination of threads. A thread runs to completion, unless a version mismatch occurs. If so, the scheduler checks if there is a currently non-running sequentially earlier thread, which is then executed. Otherwise, the thread causing the version mismatch re-executes the faulting memory access.

To explore the feasibility of the proposed versioning mechanisms, we use an FPGA-based SMP, with modified processing units in order to support versioning. The SMP is based on a template design of the GRLIB IP [1] library, chosen because of its multiprocessing support, licensing (GPL) and portability. The system consists of 8 instances of a LEON3 processor: a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture. The characteristics of this processor are shown in Table 2. The system interconnect is AMBA 2.0 AHB/APB, which is widely used as the on-chip bus for ARM processors.

ROKO is configured with two sets of parameters, shown in Table 3. The maximum numbers of LVT and GVT entries were set to the double of the largest numbers of LVT and GVT entries required by our set of benchmarks, respectively (Section 5.6). With those parameters fixed, the width of read and write version numbers are chosen so that the width of an LVT entry is 128 bits, for which the FPGA memory blocks used to store the LVT are fully utilized. The CBF is implemented as described in [11, 22]. The CBF operates on the word granularity, i.e., an entry to the CBF is added for each word of a not-acquired memory region in the LVT.

Table 2. Processor characteristics.

Processor core	1-way in-order, 6-stage pipeline 8 SPARC register windows No MUL/DIV unit, no FPU
L1 data cache	16 KB, 4-way (LRU), 32 byte blocks Write-through with no-write allocate on write-miss 1-entry (double word) write buffer
L1 instruction cache	16 KB, 4-way (LRU), 32 byte blocks Instruction burst fetch enabled

Table 3. ROKO parameters.

	Parameter	Value
LVT and GVT	The maximum number of LVT entries	128
	The maximum number of GVT entries	256
	The width of read versions	12
	The width of write versions	12
CBF	Number of counters	1024
	Number of hash functions	4
	Counter width	4

The versioning space overheads for this configuration of ROKO are summarized in Table 4. Even with our conservative estimate of the LVT size, the on-chip memory requirement is modest, totaling 15.63% of the 16KB L1 data cache.

Additional hardware and usage frequency contribute to power increase. Although a quantitative analysis of this effect is outside the scope of this work, we believe that the potential power increase is likely acceptable. The bulk of usage is the CBF accessed every data memory access, and this

Table 4. Versioning space overheads.

On-chip (per processor)	CBF	0.5 KB
	LVT	2 KB
	Total	2.5 KB
Shared memory	GVT	1 KB
	LVT (per thread)	2 KB

is no worse than a (fully-associative, accessed on every data and instruction access) TLB that usually consumes 10% of processor power.

5.2. Benchmarks

We use a set of 16 benchmarks, consisting of synthetic benchmarks, small kernels, and applications, shown in Table 5. The synthetic benchmarks are used to measure the overhead of versioning. The remaining kernels and applications present our system with a variety of data structures that exercise our programming model and hardware support. MatrixMul and MergeSort were developed by us; Search and Water are Jade [20] benchmarks, while the remaining benchmarks are taken from Olden [21].

5.3. Methodology

After the precompiler pass, the transformed source code of a benchmark is compiled and linked to the code of the ROKO run-time, producing a single executable which is then downloaded to the FPGA. We use a GNU-based cross-compilation system for LEON3 processors [1]. All benchmarks are compiled with O2 optimizations. Each experiment is repeated three times, and the average is reported.

5.4. Versioning timing overheads

Versioning introduces timing overheads because of thread management and the monitoring of memory accesses (Section 2.6). The overhead associated with monitoring memory accesses greatly depends on the application behavior, and we evaluate it in Section 5.6. In order to quantify the thread management overhead, we experimented with three synthetic benchmarks, Array, LinkedList, and BinaryTree, representing typical data structures. In these benchmarks, traversal of the data structure is divided among 8 worker threads. The breakdown of per thread overheads for different data sizes for Array and LinkedList is shown in Figure 8. The results for BinaryTree are similar to the ones for LinkedList, and are omitted for space considerations.

For Array, each thread accesses a consecutive region corresponding to one-eighth of the array. The use of variable-granularity versioning makes it only necessary to add one entry to the AT and the LVT of a child thread. Hence, the time needed to populate the AT, store the LVT and execute the release algorithm is small and independent of the size of the array. During execution of the generate algorithm and when loading the LVT of a thread, the CBF must be updated as well. Since an entry is added to the CBF for each word of a non-acquired memory region that the thread accesses, we observe a linear increase in the overhead with the array size.

Table 5. Benchmarks description and characteristics for parallel execution.

Benchmark	Data organization	Problem size	Sequential time (million cycles)	#parallel pragma	#access pragma	#threads	Avg. thread time (million cycles)	#LVT entries	#GVT entries	
Synthetic	Array	1D array	varies	1	1	9	varies	varies	varies	
	LinkedList	Single-linked list	varies	1	1	9	varies	varies	varies	
	BinaryTree	Binary tree	varies	1	1	9	varies	varies	varies	
Kernels & Applications	BiSort	Binary tree	256K integers	376	2	2	25	15.04	8	28
	Em3D	Single-linked lists, 1D arrays	10K nodes	558.46	2	4	17	32.85	16	16
	Health	Quadtree, double-linked lists	1365 villages, 200 time steps	1330.6	3	1	3001	0.44	12	66
	MatrixMul	2D arrays	400×400 integer matrix	2753.68	1	1	41	67.16	40	40
	MergeSort	1D array	32K integers	76.46	6	5	127	0.6	8	128
	MST	Single-linked lists	1K nodes	679.76	1	1	8185	0.08	17	17
	Perimeter	Quadtree	1K×1K image, 9-level quadtree	210.98	3	0	64	3.3	9	63
	Power	N-way tree, single-linked lists	1000 customers	9488.75	1	1	329	28.84	16	16
	Search	1D arrays	64 electron-solid pairs, 10 steps	4962.45	1	1	65	76.35	64	64
	TSP	Binary tree	16K cities	3035.45	1	1	8	379.43	6	14
	Union	Quad trees	1K×1K image, 8-level quadtree	27.87	3	0	64	0.44	9	63
	Voronoi	Binary tree	16K points	6120.95	1	0	8	765.12	3	7
	Water	3D arrays	343 molecules, 4 time steps	125027.84	2	8	65	1923.51	16	16

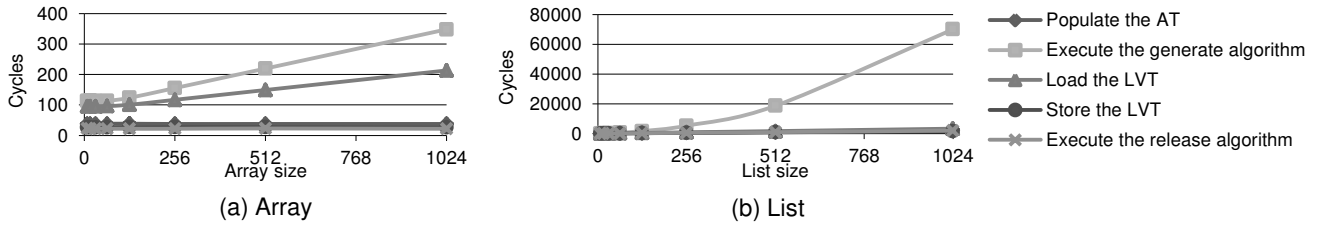


Figure 8. Thread management overhead.

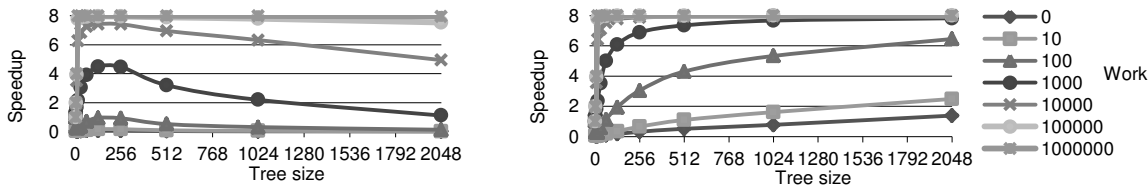


Figure 9. The performance of BinaryTree.

However, due to the hardware implementation, the growth of the overhead is small, resulting in a small overall overhead.

In contrast, a linked list is represented with multiple AT and LVT entries, one for each element of the list. Hence, time to populate the AT, load/store the LVT, and execute the release algorithm scales linearly with the size of the list. Variable-granularity versioning requires that for each LVT entry that is about to be inserted, the entire LVT is searched, in order to identify any overlaps of memory regions. The current implementation of the LVT is searched sequentially, and thus the time to execute the generate algorithm scales quadratically with the size of the list.

Consequently, when a thread traverses a large recursive structure and each access is reported as a side-effect, the thread management overheads can be significant. This overhead, however, can be mitigated by specifying only the entry point of the structure, as described earlier in Section 4. This is clearly reflected in Figure 9 which shows the speedup of BinaryTree on 8 processors, for different tree sizes and different amount of work done for each element of the tree.

In this figure, the work is simulated by a loop that repeatedly executes a `nop` instruction. The number of iterations of this loop is varied from 0 to 100000. When all accesses are reported as side-effects, a significant amount of work is needed to obtain good speedup. Moreover, performance degrades as the tree size increases, since the per-thread work scales linearly with the size of the tree while overheads scale quadratically. On the other hand, when a function’s side-effects include only accesses to the root of the subtree being traversed, good speedup is observed for a lower amount of work and it improves with the size of the tree.

5.5. Programming effort

The number of used `parallel` and `access` pragmas is shown in Table 5. Few annotations were needed to parallelize each application. Locating functions for parallelization was straightforward. It was also relatively easy to specify functions side-effects, since only the function code needed to be inspected. Identifying entry-points of recursive struc-

tures was also easy but required the inspection of the rest of the code. Nonetheless, these steps are far easier than reasoning about synchronization and deadlock.

For some benchmarks, small code changes were needed. For Water, the parallel unit of work was not at the function level, rather, it was at a finer level (two “threads” inside a single function). Hence, we extracted the corresponding code into separate functions. For Power, a global variable was used to store a temporary value in a function designated for parallel execution, which introduced unnecessary serialization. Hence, we have transformed the variable into a local one. Finally, for recursive functions annotated with `parallel` pragma, we have limited the recursion depth to which a new thread should be created, in order to avoid excessive thread creation.

5.6. Application performance

Figure 10 shows the speedups of the benchmarks on 2, 4, and 8 processors. The results are obtained for problem sizes shown earlier in Table 5. This table also presents the number of threads created during parallel execution, average thread execution time, and the largest number of entries seen in the LVT and GVT. The average thread execution time was estimated by dividing the sequential execution time by the number of threads created during parallel execution.

The geometric mean of the speedups on 8 processors for all the benchmarks is 4.09x. However, Figure 10 shows a clear distinction between 8 benchmarks for which a good speedup and scalable behavior is seen (Em3D, MatrixMul, Perimeter, Power, Search, TSP, Voronoi, Water; 5.95x speedup geomean) and 5 benchmarks for which poor speedups are obtained (BiSort, Health, MergeSort, MST and Union; 2.24x speedup geomean). To investigate the less-than-ideal speedups observed, we first examine the versioning overheads, and then examine application behavior with respect to memory performance.

The overhead of thread management is negligible over all benchmarks, contributing less than 0.4% of the total execution time on 8 processors, except for MST, for which the overhead of thread management is 1.18%. MST has a lower thread granularity, but even then, the overhead is small.

To inspect the overhead of the execution of memory accesses, we measure the delay that occurs on a CBF hit, i.e., in the cases of CBF false positives, version matches and version mismatches. We compare this time to the total execution time, as shown in Figure 11. The delay due to the CBF false positives is negligible, except for EM3D. Threads of this application access large shared arrays, leading to a large amount of memory locations held by the CBF. Consequently, the CBF makes more false positive, but these still do not significantly influence total execution time. The delay due to version matches is negligible, while the delay caused by version mismatches reaches up to 8.4% for MergeSort. This overhead, however, is due to the fact that “merge” threads repeatedly cause version mismatches while waiting for “sort” threads to finish, and it is overlapped with the work done by the “sort” threads. Thus, application performance is not degraded by versioning overheads.

The 5 benchmarks with low speedups operate on large data structures, with little computation for each element of the structure. Hence, we investigate memory contention during parallel execution. Since the bus serializes requests to the shared memory, the memory contention can lead to a significant performance degradation. We measure the percentage of the instruction causing bus traffic during the sequential execution, which signals potential memory contention during parallel execution. The results are shown in Figure 12. The 5 benchmarks with low speedups cause the most bus traffic, and thus we attribute their worse performance to contention.

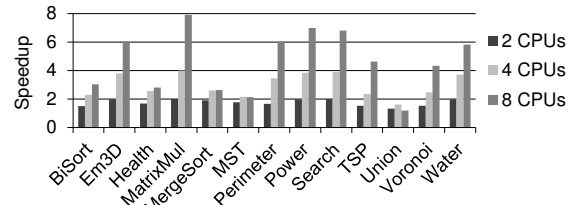


Figure 10. The speedup of the benchmarks.

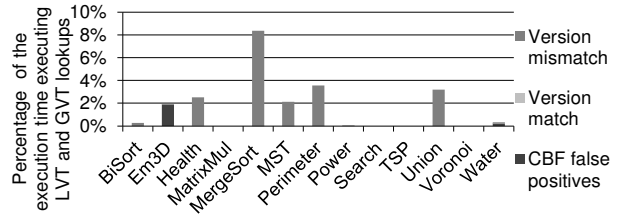


Figure 11. The time spent executing LVT and GVT lookups.

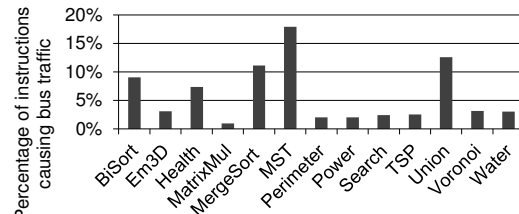


Figure 12. The bus traffic during the sequential execution

5.7. Comparison with Intel TBB

We parallelized the same benchmarks using Intel TBB 3.0 and run them on the same hardware platform, but with disabled support for versioning. Since Intel TBB requires an operating system, a port of Linux 2.6.21 [1] was run on top of our FPGA multiprocessor. Thus, the platform used to obtain TBB results was not 100% identical to the one used in the previous sections. Nevertheless, the measured sequential execution times were within 5% of the sequential times shown in Table 5, with the exceptions of Health and Union, where we notice 23.3% and 111.9% increase in the sequential execution time, respectively.

The speedups at 8 processors with ROKO and Intel TBB are shown in Figure 13. The speedups are comparable for most of the benchmarks. The greatest difference is observed for MST, where we suspect that the low speedup by TBB is

due to small amounts of work per thread (Table 5), for which the TBB overheads of thread management are not negligible. When we artificially increase the work done by each thread by 1000000 cycles, both ROKO and TBB give the same speedup (2.1x), confirming our suspicion.

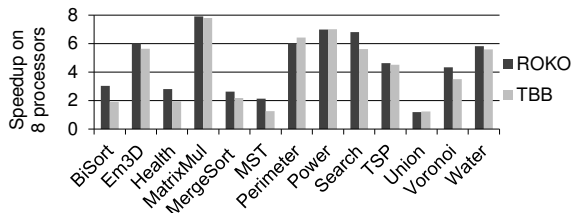


Figure 13. The speedups for ROKO and TBB on 8 processors.

6. Related work

We propose *hardware support* that induces a *programming model* in which programmers need not explicitly synchronize accesses to shared data, hence easing parallel programming. Thus, our work relates to approaches that utilize hardware support to make parallel programming easier and to relevant parallel programming models.

The multiscalar processor [23] was the first architecture to implement thread-level speculation (TLS). It requires that program instructions are grouped as tasks, and that the output registers and potential successor tasks of each task are communicated to the hardware. Using this information, the multiscalar processor, like versioning, exploits parallelism among tasks while preserving the illusion of sequential execution order. The underlying mechanisms, however, are radically different: the multiscalar processor constraints which processing unit can execute a task and uses data and control speculation to predict memory dependences and the successor task. In contrast, versioning puts no restrictions on thread-to-processor mapping, and employs no speculation. Later TLS architectures [24] trade some performance for simplicity, but rely on software optimizations (e.g., compiler-inserted synchronization [25]) to improve performance.

TM replaces locks with transactions, but does require that critical sections are identified and made into transactions. It also employs a speculative, and moreover, a non-deterministic execution model. Our non-speculative hardware by design avoids some of the issues of (speculative) TLS and TM architectures (e.g., wasted power of non-commits and difficulties in parallelizing code with I/O and system calls).

Colorama [6] is an example of a *non-speculative* hardware support that addresses the issues of parallel programming. With Colorama [6], programmers partition the data and the hardware support triggers the start of a critical section upon the first access to a partition. Colorama uses a non-deterministic execution model and can cause deadlocks when locks are used as the underlying synchronization mechanism.

OpenMP [7], Intel TBB [19], Cilk [14] and Cilk++ [16] are popular software approaches for writing multithreaded

programs on commodity hardware. Although effective, these approaches require programmers to *explicitly* synchronize accesses to shared data and may lead to non-deterministic execution. Synchronization is one of the more difficult aspects of parallel programming and versioning induces a synchronization-free deterministic programming model, albeit at the cost of specifying data accesses.

More closely related to our work are programming models with *implicit* synchronization [2, 4, 5, 20]. Prometheus [2] is a C++ library that implements a deterministic execution model using the notion of *serializers*. To ensure valid execution, programmers must write serializers to dynamically inspect data accessed by a function and establish a serialization set for the function. Multiple functions in the same serialization set execute serially on the same processor. With versioning, programmers only add accessed data, which the system uses to dynamically serialize code when necessary. This simplifies parallelization albeit at the cost of extra hardware. Furthermore, Prometheus does not allow overlap between execution of function calls that access the same data, nor does it support nested parallelism, while versioning does allow for both.

In Jade [20], programmers summarize data accesses, with synchronization being implicit and deterministic. However, compared to versioning, Jade puts the additional burden of annotating declarations of shared data and marking certain accesses to it. Also, the granularity at which synchronization is enforced is determined when shared objects are declared, and it cannot be changed during run-time, unlike our variable-granularity versioning. Grace [4] executes function calls speculatively in parallel, and in separate address spaces. After a function call is executed, writes are merged to the main address space in the sequential order, but only if no conflict occurs. Otherwise, the work is squashed and repeated. Grace requires no hardware support; however, its conflict detection works at page granularity, requiring significant code modifications to minimize false sharing.

Deterministic Parallel Java (DPJ) [5], similar to our work, uses programmer-provided data access summaries to guarantee determinism during parallel execution of Java codes with serial semantics. DPJ allows for the parallel execution of arbitrary blocks of code, as long as it can be proven that these blocks of code have no conflicting data accesses. Conflicting accesses are detected through a type and effect system that allows programmers to partition the heap into named regions with associated compiler analysis to statically determine data dependences. In contrast, versioning detects and enforces dependences at run time and at the level of memory locations, which has several advantages. First, it provides a synchronization mechanism that is divorced from the programming language and independent from potentially limited compiler analysis. This allows the use of versioning with common languages, strongly typed or otherwise. Second, versioning leads to a slightly simpler programming model than DPJ since versioning demands only the specification of data access summary while DPJ demands both the partitioning of the heap and the specification of data access summaries. Finally, the enforcement of statically-determined dependencies does not permit overlap among ex-

ecution of code blocks with conflicting access, but versioning does allow for such overlap, without sacrificing determinism. Nonetheless, DPJ requires no hardware support and its constructs allow for explicit controlled nondeterminism when such behavior is desirable.

Recently, there have been a few proposals for supporting deterministic execution of otherwise non-deterministic, explicitly synchronized codes using transparent runtime techniques [10, 9, 18]. However, the order of memory accesses during a parallel execution, although deterministic, is unknown to the programmer. Versioning ensures that the order of accesses is equivalent to the one in serial execution.

Our work also relates to the hardware proposals for fine-grained monitoring of memory accesses [3, 26, 28]. Compared to our monitoring mechanism of a Bloom filter, these systems either require the use of a ternary content addressable memory [26, 28], which is more expensive [13], or demand changes to the memory hierarchy [3].

7. Conclusion and future work

We propose a novel synchronization scheme called versioning that provides deterministic execution for parallel programs with serial semantics and alleviates the need for programmers to explicitly synchronize threads. Architectural support to make versioning efficient consists of dedicated on-chip storage and a small coprocessor, and does not require changes to the processor pipeline, to the caches, nor to the coherence protocol. This support automatically and dynamically detects and enforces dependences among parallel threads, inducing a parallel programming model in which there is no explicit synchronization of threads and thus no need to deal with atomicity or mutual exclusion.

We built an FPGA prototype of an 8-processor system with support for versioning and evaluated the performance of 13 standard benchmarks parallelized using a proposed task-level programming model, similar to Cilk and Cilk++. We find that with our architectural support, versioning overheads do not have significant impact on the application performance. The geometric mean of the speedups on 8 processors is 4.09x for all benchmarks, and 5.95x for 8 applications that are not memory-bound. Thus, we conclude that versioning delivers good performance with ease of programming.

This work can be extended in several directions. Compiler analysis can be used to filter data accesses monitored at run-time, reducing overheads, and also to assist programmers report functions' side-effects. Further, the effect of the additional versioning hardware on processor power can be quantified. Finally, a scalability study can be performed to study versioning with more than 8 processors and with more applications.

References

[1] Aeroflex Gaisler. <http://www.gaisler.com/>.
[2] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization Sets: A Dynamic Dependence-Passed Parallel Execution Model. In *Proc. of PPOPP*, pages 85–96, 2009.

[3] L. Baugh et al. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. In *Proc. of ISCA*, pages 115–126, 2008.
[4] E. D. Berger et al. Grace: Safe Multithreaded Programming for C/C++. In *Proc. of OOPSLA*, pages 81–96, 2009.
[5] R. L. Bocchino, Jr. et al. A Type and Effect System for Deterministic Parallel Java. In *Proc. of OOPSLA*, pages 97–116, 2009.
[6] L. Ceze, P. Montesinos, C. von Praun, and J. Torrellas. Col-orama: Architectural Support for Data-Centric Synchronization. In *Proc. of HPCA*, pages 133–144, 2007.
[7] B. Chapman et al. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.
[8] L. Chris. LLVM and Clang: Next Generation Compiler Technology. In *Proc. of BSDCan: The BSD Conf.*, 2008.
[9] J. Devietti et al. RCDC: A Relaxed Consistency Deterministic Computer. In *Proc. of ASPLOS*, pages 67–78, 2011.
[10] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *Proc. of ASPLOS*, pages 85–96, 2009.
[11] S. Dharmapurikar et al. Deep Packet Inspection using Parallel Bloom Filters. *IEEE Micro*, 24(1):52–61, 2004.
[12] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE Trans. Netw.*, 8(3):281–293, 2000.
[13] Y.-T. Fang et al. Ternary CAM Compaction for IP Address Lookup. In *Proc. of AINAW*, pages 1462–1467, 2008.
[14] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proc. of PLDI*, pages 212–223, 1998.
[15] M. Herlihy and J. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. of ISCA*, pages 289–300, 1993.
[16] C. E. Leiserson. The Cilk++ Concurrency Platform. In *Proc. of DAC*, pages 522–527, 2009.
[17] S. Lewin-Berlin. Four Reasons Why Parallel Programs Should Have Serial Semantics. <http://software.intel.com/en-us/articles/four-reasons-why-parallel-programs-should-have-serial-semantics>, 2009.
[18] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *Proc. of ASPLOS*, pages 97–108, 2009.
[19] J. Reinders. *Intel Threading Building Blocks*. O'Reilly, 2007.
[20] M. C. Rinard and M. S. Lam. The Design, Implementation, and Evaluation of Jade. *ACM Trans. on Prog. Lang. and Systems*, 20:483–545, May 1998.
[21] A. Rogers et al. Supporting Dynamic Data Structures on Distributed-Memory Machines. *ACM Trans. on Prog. Lang. and Systems*, 17(2):233–263, 1995.
[22] D. Sanchez et al. Implementing Signatures for Transactional Memory. In *Proc. of MICRO*, pages 123–133, 2007.
[23] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proc. of ISCA*, pages 414–425, 1995.
[24] J. G. Steffan et al. A Scalable Approach to Thread-Level Speculation. In *Proc. of ISCA*, pages 1–12, 2000.
[25] J. G. Steffan et al. Improving Value Communication for Thread-Level Speculation. In *Proc. of HPCA*, pages 65–, 2002.
[26] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proc. of ASPLOS*, pages 304–316, 2002.
[27] L. Yen et al. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *Proc. of HPCA*, pages 261–272, 2007.
[28] P. Zhou et al. iWatcher: Efficient Architectural Support for Software Debugging. In *Proc. of ISCA*, pages 224–, 2004.