2008 International Symposium on Parallel and Distributed Processing with Applications

# Parallelisation of the Valgrind Dynamic Binary Instrumentation Framework

Daniel Robson and Peter Strazdins
Department of Computer Science
The Australian National University
{drobson,peter}@cs.anu.edu.au

## Abstract

*Valgrind is a dynamic binary translation and instrumentation framework. It is suited to analysing memory usage. It is used in memory validation and profiling tools. Currently, Valgrind is restricted to executing a guest with serialised thread scheduling. This results in lost opportunity for performance when analysing highly parallel applications on parallel architectures.*

*We have extended the framework to allow parallel execution of guest threads. Code caching mechanisms have been made thread-safe, by delaying flushing of translated code, while preserving critical areas of performance.*

*Three methods which preserve atomicity of instructions are implemented and evaluated with respect to speed, reliability and instrumentation effects. Serialising both store and atomic operations preserves atomicity in the strongest sense, but suffers unacceptable performance overhead. Serialising only atomic instructions or utilising host atomic instructions provides speedup in line with native execution. These methods show average slowdowns of only 2.6× and 2.2× over native parallel execution respectively.*

## 1 Introduction

Binary translation forms a crucial component of many validation [12], simulation [2, 3] and software performance [7] tools. This involves translating instructions from their target Instruction Set Architecture (*ISA*) to a secondary ISA, perhaps through intermediate modification stages[1]. The translation process provides an opportunity to modify the semantics of instructions, or add instrumentation.

Performance evaluation tools may use features of the underlying code, such as memory references, which are passed to analysis routines as part of modified execution. This allows for an in depth view of an application's execution while maintaining reasonable performance. We are primarily interested in analysis of large, parallel, scientific workloads due to their strong reliance on CPU and memory performance.

*Valgrind* [9] is a dynamic binary translation and instrumentation framework which is suited to analysing memory usage on x86, AMD-64 and PPC architectures. It provides facilities for loading, translating and executing an application (the *guest*) without linking or source modification requirements. For a tool developer, it provides a layer exposing code instrumentation and event notification mechanisms.

However the current implementation prevents parallel execution of guest threads. As many scientific workloads are designed for large numbers of processors, serialisation represents a lost opportunity in execution speed as compared with native parallel execution. Additionally, serialisation makes applying timing sensitive instrumentation difficult.

We have modified the Valgrind framework, described in detail in section 2, to operate with multiple concurrent guest threads (referred to as *pValgrind*), with the AMD-64 architecture of chief interest. In order to facilitate multithreaded execution, two key deficiencies were addressed.

In section 3, we describe modifications to translation caching and flushing required to retain safety and performance. In section 4, we describe three modifications to disassembly and execution of atomic instructions which retain atomic semantics and instrumentation capabilities. The performance of each method is discussed in section 5.

Two tools have been extended to operate concurrently, *Lackey* and *Cachegrind* (referred to as *pLackey* and *pCachegrind* respectively), which we describe in section 6. Both show increased performance with multiple CPU systems.

Section 7 relates our approach to existing work while section 8 concludes.

## 2 Background

The Valgrind framework is divided into three main areas: core and guest maintenance (*coregrind*), translation and instrumentation (*LibVEX*), and user instrumentation libraries.

[1] Original, final and intermediate ISA used in translation need not be unique.

978-0-7695-3471-8/08 $25.00 © 2008 IEEE
DOI 10.1109/ISPA.2008.94

```
0x4001D5E:  subq 2199099(%rip),%rax

------ IMark(0x4001D5E, 7) ------
PUT(168) = 0x4001D5E:I64
t22 = Add64(0x4001D65:I64,0x218E3B:I64)
t21 = GET:I64(0)
t20 = LDle:I64(t22)
t19 = Sub64(t21,t20)
PUT(128) = 0x8:I64
PUT(136) = t21
PUT(144) = t20
PUT(0) = t19
```

**Figure 1. Components of an AMD-64 to UCode translation of a subtraction from indexed memory**

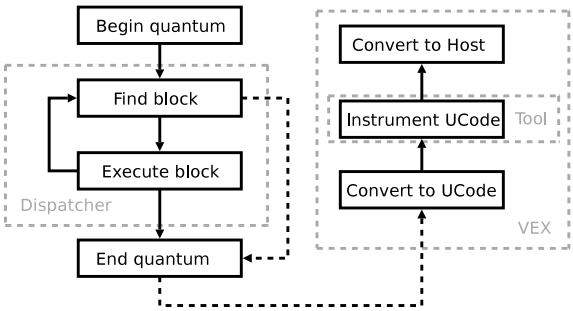


**Figure 2. Valgrind translation, instrumentation and dispatch loop**

All three elements work in conjunction to load, instrument, and execute a guest binary. The combination of these elements is referred to as a Valgrind *tool*.

Coregrind contains OS specific routines emulating the normal system image loading and startup methods. Using these, a tool will load and position a guest within its own address space. As a guest runs, all code is passed through LibVEX binary translation prior to execution. With the careful manipulation of a guest's translation, all scenarios where a guest could regain direct control of execution are eliminated.

A tool may dynamically modify the instructions of a guest at translation time. Manipulation is performed through a specialised RISC-like intermediary ISA, *UCode*. The original ISA is translated to UCode, modified and instrumented, and subsequently translated to the host ISA. UCode is designed for simple access of memory addresses and instruction sub-operations, avoiding details that may be complex to decode and account for in a CISC architecture.

Each guest instruction may be potentially broken into a number of UCode operations, such as calculating addresses or intermediate steps of the instruction. Figure 1 illustrates the operations performing a subtraction on memory. After tool instrumentation, an instruction could consist of an arbitrary number of operations, typically involving a call to a tool function which records statistics or modifies a model of state.

Translation and execution of a guest is performed at the basic block scope[2]. Each basic block executes from its start address through to a terminating jump to a predefined Valgrind supplied function, the *dispatcher*. This small, assembly function is responsible for calculating and jumping to the next basic block; iterating through each guest basic block in turn, as outlined in figure 2.

Each translated block is cached in a large global structure so as to reduce re-translation overhead. As basic blocks do not directly jump to their successor, the overhead of block lookup must be minimized to ensure adequate performance[3]. As the global cache is optimised for efficient address range flushing and management, performing this lookup on the global cache is expensive. Instead, a fast direct mapped table of addresses to translated block pointers is used, exploiting locality of reference.

The current scheduling of guest threads is designed to simplify the architecture of Valgrind and execution of guest code. Only a single thread is permitted to execute userspace code at a time; a single global lock must be held to execute any userspace code, whether in Valgrind or guest. Each thread will execute a maximum quantum of basic blocks (currently 10,000) within the dispatcher before deferring to other threads.

Threads yield their quantum prior to executing syscalls, when requiring untranslated code blocks or when a variety of guest faults occur. After dropping out of the dispatcher to yield, a thread will perform some routine guest maintenance: execution of queued signals and periodic sanity checks. When a syscall is requested the thread will potentially execute in parallel with userspace code, re-acquiring the execution lock prior immediately it returns from kernelspace.

There is no attempt to synchronise thread execution based on notions of time or fairness. Complete usage of execution quantum is not guaranteed and will vary considerably due to placement of syscalls, efficiency of translation caches and other execution characteristics. Due to variable length translation and code instrumentation for each instruction, even a single thread's execution in isolation can not be guaranteed to reflect the corresponding native tim-

[2]A single run of instructions which contain a single entry and possibly multiple exit points.

[3]The performance of the Strata dynamic binary translation framework increased 5 fold when block chaining was introduced, due to a 250 cycle block lookup [9, 11]

ings.

## 3 Parallelisation issues in translation caching

A key difficulty in efficient guest execution within Valgrind's translation infrastructure is to ensure that the lookup and execution of a translated block are performed sufficiently quickly. The dispatcher inner loop for the AMD-64 architecture consists of 14 instructions, with the fast table lookup accounting for half. As there is typically 5-30 guest instructions per basic block[4], these operations must be performed quite frequently.

The global translation cache is of finite size and flushes can occur under a number of scenarios: when the cache becomes greater than 80% full, when a tool requests an address range flush, and when self modifying code is executed.

This presents a problem when a thread is currently executing a block to be flushed. In serial Valgrind only the single executing thread has the ability to delete blocks. This thread could not be in a translated block as all cache manipulation occurs outside of guest execution. With multiple executing threads, the block cannot necessarily be freed in this scenario as its code could potentially be in use. As recording whether a block is being executed or locking table entries would cause prohibitive performance penalties, there is no efficient mechanism to determine whether a block is being executed.

We have introduced a list of blocks pending deletion. This allows threads to continue executing currently referenced blocks while permitting efficient block flushing and guest execution.

As the cache manipulation routines potentially use the LibVEX translation library, which is not thread-safe, we introduced a lock to protect all access to these routines. The only operation on the translated cache which does not require a lock is a read from the fast table.

There are two locations which could contain pointers to translated code blocks: the global cache and the fast translation tables. The translation to be flushed is first removed from the global cache, so that it cannot be reintroduced into the fast table. Any thread requesting a mapping to this address will trigger a re-translation.

The thread performing the flush then atomically substitutes a null pointer into the fast table if it contains this block. A null pointer in a fast table is interpreted as having no known translation and triggers a translation request when next accessed. As we are not modifying the block's data, a thread currently executing this block can continue safely.

The flushed translation is placed in the list of blocks pending deletion until we can safely determine it is not in use. A bitmap indicating which threads currently executing at the time of flushing is created and stored with the block. As part of normal thread maintenance at the end of the execution quantum, each thread iterates through all flushed translations and zeroes their corresponding flag. This indicates that a thread has no live reference to this translation and is not capable of re-acquiring access to it. When the entire bitmap is cleared, the block is finally freed.

[4]We have observed 19 guest instructions per block on average in the NPB W class

| Time | Thread | Thread B |
|---|---|---|
| 1 | `t5 = LDle:I32(t6)` | |
| 2 | `t7 = Sub32(t5,0x1:I32)` | |
| 3 | | `t5 = LDle:I32(t6)` |
| 4 | `STle(t6) = t7` | |
| 5 | | `t7 = Sub32(t5,0x1:I32)` |
| 6 | | `STle(t6) = t7` |

**Figure 3. Overlapping atomic decrement in UCode with two threads**

## 4 Preservation of instruction atomicity

With any system which breaks instructions into multiple operations, care must be taken to avoid introducing memory ordering problems, particularly with atomic instructions. As shown in figure 3, there is a possibility of overlapping instruction (sub-) operations. In the case of Valgrind, these operations are translated into multiple host instructions, destroying their atomicity if executed in parallel. The problem is compounded by the possibility that a tool could modify the disassembled instruction in an arbitrary manner, typically adding further instructions and creating a larger window for errors.

Valgrind's serialised thread scheduling of userspace guest code avoids the issue of instruction atomicity. All threads execute basic blocks to completion before attending to any pending events, and any signals received are queued for later delivery. This ensures no thread can be interrupted during the execution of a basic block, and hence of an atomic instruction. Combined with thread serialisation, all atomicity hazards are eliminated, at the cost of parallelism.

Methods used to preserve instruction atomicity must impose minimal overhead when the majority of instructions are executed (it should be noted that atomic instructions generally constitute a small fraction of the total executed). Owing to the nature of instruction disassembly and reassembly, it is difficult to retain both the expressive power of UCode and functional semantics of atomic instructions at the instruction level. An ideal method would preserve all generated UCode.

We have developed three methods of atomicity preservation. Each has various safety and performance characteristics with differing impact on tool development and maintenance. The implementation of each method is compared with the original thread serialisation approach. The perfor-

mance of each is outlined in section 5.

### 4.1 Store and atomic serialisation by locking

By acquiring a lock before performing a store or atomic instruction, we can enforce the atomicity of all store operations. This provides substantially stronger atomicity guarantees within the guest than native execution.

Our implementation uses an array of $2^{n+1}$ spinlocks padded to cacheline length. Bits $[n+6:6]$ of the atomic instruction's virtual address are used to index this array. During the initial translation of each guest instruction, pValgrind scans for store operations. Immediately prior to each store operation, a function call to acquire the appropriate lock is introduced, together with a corresponding call to release the lock at the end of translation. If any load operation to the same address was found within the translation prior to the store, the lock acquisition call is placed immediately before the load instead. This ensures the serialization of potential atomic instructions.

### 4.2 Atomic serialisation by locking

A potential drawback of the above approach is that it imposes overhead on all store operations. Because of this, we examined two methods involving only the serialisation of atomic instructions. The first is based on the supposition that if a 'well behaved' application manipulates a memory location with an atomic instruction, it is likely that only atomic instructions would be writing to that address at that time. This method is described below, with the second method being described in section 4.3.

Known implementations of synchronisation primitives which utilise atomic instructions (e.g. a barrier with an atomic decrement) manipulate their data exclusively with atomic instructions. If a thread was waiting on a barrier variable to be (atomically) decremented to 0, and another thread wrote with a non-atomic operation, the synchronisation could be lost. Thus standard synchronisation methods can be performed safely when atomic instructions are serialised.

A single globally accessible spinlock was introduced which must be held for the duration of any atomic instruction. Functionally, this method is equivalent to the method of section 4.1 with a single lock, but applied only to atomic instructions. Thus, its implementation, portability and protection concerns have similar characteristics, with a potential reduction in overhead.

### 4.3 Atomic serialisation with host atomics

To support precise atomic instruction semantics, a final method was developed which utilises the guest instruction in question, or a functionally equivalent host instruction. The initial conversion from guest instructions to UCode was modified to emit a call to a function; address calculation operations were still emitted.

```
/* Given the host register and state
 * information we are to increment
 * address 'addr' of size 'ty' for the
 * guest. No side effects other than
 * memory
 */

static
void atomic_c_INC(
      VexGuestAMD64State * guest,
      IRType ty,
      HWord addr) {
  ULong flags;

  switch(ty) {
    case Ity_I8:
      __asm__ __volatile__ (
        "LOCK incb (%%rax);\n"
        "pushf;\n"
        "popq %%rbx;\n"
        : "=b"(flags)
        : "a" (addr)
      ); break;

      // Similarly for other sizes...
```

**Figure 4. Host atomic execution of an atomic increment**

The instruction's operands are passed to the function in addition to the current thread state (which includes all register and CPU state). This function contains inline assembler for the instruction in question, in the style of figure 4. After the instruction has been executed, the relevant CPU flags are saved into the guest state and results are returned if appropriate.

This method is implemented for the AMD-64 architecture only, which has 19 atomic instructions.

### 4.4 Discussion

All methods solve instruction atomicity issues, each with various implementation and reliability concerns. Some methods are potentially sensitive to UCode transformation, some provide more UCode detail and there are different levels of reliability.

While all methods preserve atomic semantics within the guest's userspace components, guest memory accesses made by the kernel or other external agents need not follow the same protocol. This can result in the loss of atomicity when using the locking based methods, but not with the host atomics method. Note that this problem is present in current versions of Valgrind, but is not known to have caused substantial problems so far.

| Time | Thread A (atomic) | Thread B (non-atomic) |
|---|---|---|
| 1 | READ [m1], r1 | |
| 2 | | WRITE [m1], r2 |
| 3 | INC r1 | |
| 4 | WRITE [m1], r1 | *Lost previous write* |

**Figure 5. Serialised atomics: Write loss with atomic increment and parallel non-atomic store**

All methods place some limitations on LibVEX's optimisation phases, as embedded function calls marked as touching guest state or memory require cached guest state (i.e. registers) to be written out prior to the call. This ensures a coherent view of guest state, but limits the location and duration of block level optimisations. This problem is exacerbated within the host atomic method, when CPU flags must be saved. This requires additional memory operations that lessen LibVEX's capacity to cache flag values.

The methods presented can be applied to any architecture. Serialised stores and serialised atomics are most readily transferable due to the fact that all instruction operations are expressed in UCode. However, as both methods rely on strict positioning of locks, they have the potential to fail under certain patterns of UCode transformation or reordering[5]. This does not apply to the host atomics method, which, due to the fact that all crucial guest state updates are carried out as a single atomic operation, is inherently more robust to UCode transformations.

Host atomics require an implementation of each guest atomic instruction with a corresponding host atomic instruction, for each combination of host and guest ISA to be supported, resulting in a programming burden. Despite the reduction in UCode detail in host atomics, we do not believe that tool functionality will suffer with this mechanism, as the relevant memory references are still made available.

It is unclear whether the serialised atomics method would result in reliability problems for practical applications. There are potential issues with simultaneous non-atomic write loss, as in figure 5. We have not observed any instances where this causes problems, particularly in benchmarks used in section 5. From a practical standpoint, if this mechanism proved problematic for a specific application, other atomic preservation methods could be utilised, including reverting to thread serialisation.

## 5 Performance

To establish the efficiency of our approaches we have analysed the performance of the NAS Parallel Benchmarks, *NPB* [1]; a set of 5 kernels and 3 applications constructed from computational fluid dynamics applications. We have

[5]We are not aware of any existing tools which modify instruction or UCode ordering in such a manner.

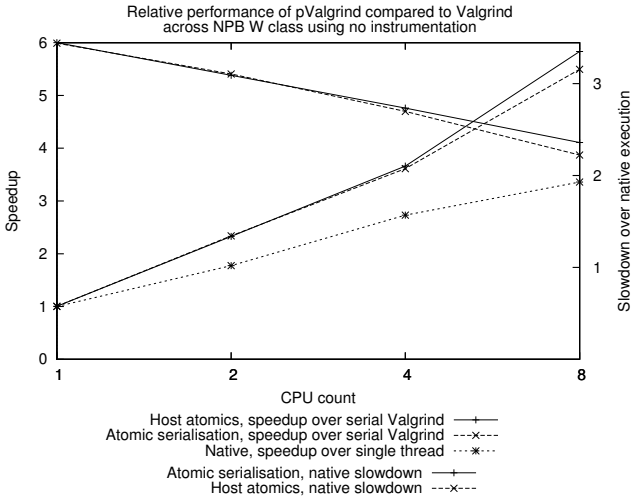


**Figure 6. Geometric mean of multiprocessing performance of atomic serialisation methods on NPB (W class) with no instrumentation.**

chosen to analyse version 3.0 of the OpenMP reference implementation. Each benchmark of the W class was run 5 times; the average execution time of the benchmark's computational phase for further analysis. A geometric mean on these results was used to get an overall result for the NAS.

Valgrind, pValgrind and NPB were all compiled with GCC 4.2.3 for all tests. Valgrind and pValgrind were built with only 64 bit support using the standard configure/make procedure; the process implicitly builds all default tools, e.g. Lackey and Cachegrind. NPB was compiled with level 1 optimisations enabled.

All performance tests were conducted on a SunFire X4600 M2 unless otherwise noted; an eight-socket, dual-core Opteron 8128 system operating at 2.67 GHz, with a total 32 GiB of system RAM. The system was running RedHat Linux, kernel 2.6.18.

All comparisons to serial Valgrind were conducted with version 3.3.0. Comparisons of workload times were performed without instrumentation, using the *Nullgrind* tool, to determine the basic performance penalty of each parallelisation method. Similar comparisons for instrumented execution are given in section 6.3. All analysis of native, pValgrind and Valgrind timings were compared against configurations utilising the same number of threads.

The performance of pValgrind using the Valgrind's serialised threads scheduling compared with Valgrind is almost identical. The overhead of storing additional state such as TIDs and locking of cache structures shows less than 1% performance difference. In both cases there is a linear slowdown of execution with the addition of hardware supported threads as compared to the native execution.

Both atomic serialisation and host atomics, in all tools,

see a clear performance improvement with multiple CPUs, as compared with serial Valgrind (running with the same number of threads), as illustrated by figure 6. This indicates the performance gains achieved by utilizing pValgrind.

A geometric mean slowdown over native execution of between 3.45 and 2.36× for serialised atomics and 3.43 and 2.22× for host atomics was observed when tested over 1 to 8 CPUs. The fact that the slowdown decreases with the number of CPU indicates that pValgrind scales correspondingly better than does native execution. This is primarily due to the fact that the overheads introduced by pValgrind (i.e. the maintenance of guest processor state) dilutes the strain on the memory system of the original application.

The serialised store and atomic method was observed to have substantial performance loss, with slowdowns between 1.63× and 60× (with a geometric mean of 9×) over Valgrind. For this reason, it could not be plotted on figure 6.

Performance results for 16 CPUs as not discussed, as the NAS benchmarks did not scale beyond 8 CPUs under native execution. This even includes the EP benchmark which generates no coherency traffic. This effect is due to the fact that the memory-intensive nature of these applications saturated the memory system, coupled with the fact that the unmodified benchmarks do not try to optimize memory placement on a host with strong NUMA effects.

### 5.1 Store and atomic serialisation by locking

Figure 7 indicates the performance of the store and atomic serialisation method when varying the number of locks within $8 \leq n \leq 4096$, with four threads on four CPUs without instrumentation. The analysis was performed on a quad-core Intel Q6700 (clocked at 2.67 GHz) with 4 GiB RAM running Linux 2.6.24. All results showed performance decreased as the number of locks dropped from 1024, as in figure 7. Lock contention had significant impact with $n \leq 128$, where slowdowns reached over 3.2× with the BT benchmark.

There were diminishing returns for increased numbers of locks where $n \geq 1024$, with a maximum of 4% difference in performance beyond this point. All further performance evaluations were performed with 1024 locks.

A geometric mean slowdown of 9× for the NPB was observed over all other Valgrind implementations in the single threaded case, including thread serialisation. With increased processors, the slowdown over serial Valgrind dipped to 7.5×.

### 5.2 Discussion

Modifying many operations in a translation shows serious performance drawbacks. While serialising stores satisfies goals for preserving atomicity, the performance hit is typically significantly above any observed useful functional gain, with less performance than serialised threading solutions utilised by serial Valgrind.

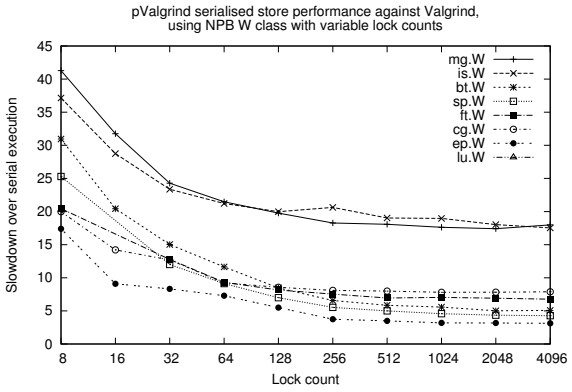


**Figure 7. Performance of store serialisation without instrumentation on the NAS Benchmarks, relative to 4096 locks. Executed with 4 threads on 4 CPUs.**

The performance of atomic serialisation and host atomics show the benefit of modifying only atomic instructions, which occur infrequently in typical applications. Both methods show speedups at least in line with native and are bound primarily by the efficiency of generated code.

Host atomics scale slightly better than atomic serialisation. Both methods use a function call and at least one atomic operation and conditional jump. However serialised atomics has lower worst case performance for the case where many threads compete for the lock, potentially introducing many more coherency events. This contributes to reduced performance with 8 or greater CPUs.

## 6 Tool parallelisation

To establish the utility of our approach, we need to demonstrate that performance analysis tools can be easily adapted for pValgrind, and show similar performance benefits. We chose to parallelise two tools distributed and built with Valgrind, Lackey and Cachegrind, to this effect. In developing these extensions, we discovered some common implementation considerations in addition to those of serial Valgrind.

Many tools incorporate structures which are updated directly by the current thread. Modifying these structures with multiple threads can cause heavy cacheline contention on the host, as updates are typically performed very frequently. A default quantum of 10,000 basic blocks hides contention problems from serialized tools. Where feasible, extending data structures such that statistics are kept per-thread and merged at finalisation can solve this problem.

Indexing state for thread parallel data structures by thread ID caused some performance degradation as there was no efficient mechanism to query the thread context. Serial Valgrind contained the notion of a currently executing thread, which was stored in a globally accessible variable.

Using a syscall to get the thread ID in every event to index state was too slow. As this is commonly required data, we introduced an additional element into the Valgrind/LibVEX thread-state structures. A tool can request the TID at any point in precisely the same manner as examining any register, typically adding it as a parameter to a tool function call. The only penalty when the element is not used is that of storing its value immediately prior to entering the dispatcher.

A substantial amount of consideration has gone into making Valgrind an effective shadow memory tool [9]. Under pValgrind, a guest's memory space can potentially be manipulated by any other running thread, making a coherent shadow memory implementation difficult. Parallelising shadow memory tools thus remains an open problem, with Memcheck still requiring serialised scheduling.

### 6.1 Instruction profiler

An advanced instruction profiler, *Lackey*, is provided with the Valgrind distribution. It is capable of keeping tallies of all instructions, branches and memory accesses attributable to various classes of guest instructions. Lackey was chosen to parallelise as its simple tasks allow for an easy implementation and should result in reasonable performance increases.

Parallelisation was achieved through a simple modification to use per-thread event counter structures which were summed at tool finalisation. An initial naive implementation involving shared counters and atomic modification highlighted the severity of poorly chosen data structures, with over 100× lower performance over serialised threads with four CPUs.

### 6.2 Cache profiler

A cache profiler, *Cachegrind*, forms part of the default Valgrind tools. It contains a model of a single two-level cache with parametrised capacity, associativity and line size. As there is a single cache, each thread takes turns updating the model during their quantum. While not modelling a multiprocessor system, it serves as a useful approximation, particularly of an n-core shared L2 design.

A simple modification to Cachegrind was performed which provided a separate simulated two-level cache for each thread. A thread thus becomes an approximation of a processor without shared caches, such as the AMD Opteron. To simplify implementation, the model does not account for coherency operations. This does not affect functional operation. However for applications with frequently shared data, results will be optimistic due to the same lines existing in multiple caches simultaneously.

The cache configuration modelled for performance comparison was that of the benchmark host: 64 KiB L1, 2-way associative with 64B lines and a 1 MiB L2, 8-way associative with 64B lines.

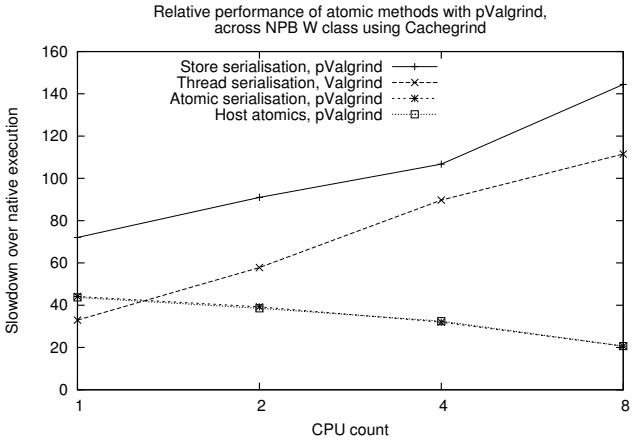


**Figure 8. Geometric mean of pCachegrind performance on NPB W class compared to serialised tools**

### 6.3 Performance

Figure 8 shows slowdowns over native execution for both pCachegrind[6]. Again, the fact that the slowdown does not increase with CPUs indicates that the host atomics and atomic serialization methods scale well. In fact, the slowdowns are reduced by half as CPUs increase from one to eight, even more than in the uninstrumented case (figure 6). This indicates a speedup at 8 CPUs of twice that of native execution for the NAS.

As with the no-instrumentation case, this effect can be explained by the even greater degree of overhead of the tool diluting the memory system load of the original application. It should be noted that within the tool, the communication between threads is limited to the finalisation of an application.

A comparison of the 1 CPU performance on figure 8 indicates that here Valgrind is slightly faster. This reflects the overheads of using arrays of guest state models, which necessitates indexing and hence additional memory indirection.

Both serialised threads and serialised stores saw an increasing slowdown with the number of CPU, indicating poor scaling behavior. As the latter method fails to attain performance improvement of serial Valgind even in the instrumented case, it looks doubtful whether it is likely to be worthwhile in any context.

## 7 Related work

While a number of parallelised frameworks exist which perform similar operations to Valgrind, there is a lack of discussion on the techniques utilised or problems encountered

[6] pLackey shows extremely similar performance, with relative 8 CPU performance of 160× for store serialisation

on parallelization. The frameworks mentioned below lack this and also do not offer any analysis of parallel execution.

Embra [13] pioneered the usage of parallel execution of translated code, with both serialised round-robin and parallel execution. Its caching and dispatch method is similar to that of serial Valgrind, while also implementing block chaining. Valgrind differs with its focus on providing a base to perform instrumentation rather than simulation, and as such must provide far more general methods for instrumentation and instruction modification.

The Embra authors do not discuss the implementation of atomic instructions or many parallelisation issues. The instruction sets targetted by Embra, the MIPS R3000 and R4000, include fewer atomic instructions than the AMD-64. Inspection of the Embra source code shows they use a method similar to our host atomics instructions method when using parallel execution, while decomposing atomic instructions when using serial execution.

There are a number of dynamic instrumentation frameworks available for similar platforms to Valgrind. Pin [8] and DynamoRIO [4] are popular JIT based systems with parallel execution. As both utilise a copy-and-annotate approach, where most instructions are copied unmodified to cache, they avoid atomicity concerns in re-assembly. They are more efficient than Valgrind when performing lightweight analysis.

DynInst [5] and DTrace [6] are popular probe based systems with parallel execution, which are not intended to be utilised for extremely fine grained instrumentation. They permit function calls to be inserted and only modify sufficient guest state to gain control of execution at the appropriate time. This class of tool typically patch guest instructions at runtime, making caching and atomicity concerns irrelevant.

Valgrind's primary advantages lie in the use of UCode. UCode is substantially easier to analyse than complex instruction sets like AMD-64, and provides a platform neutral abstraction layer over all instruction sets[7]. This assists tools, such as Memcheck, which rely on analysing implicit operations and values in complex instructions. As UCode is necessarily as expressive as host code, both tool and analysis code is similarly as expressive.

Valgrind's code optimisation routines treat guest and tool codes equally and can thus more tightly optimise host code generation, increasing the performance of a complex instrumentation. In comparison, Pin attempts to inline instrumentation code, but is prevented from doing so if any flow control statements are present.

[7] At time of writing Valgrind supports bi-directional conversion between UCode and the x86, AMD-64 and PPC instruction sets

# 8 Conclusions

With a small number of additions to the core Valgrind framework, we are able to efficiently parallelize the instrumented execution of threaded applications. We have demonstrated a practical method of parallelisation of the Valgrind framework.

The use of a list of blocks for delayed flushing retains the performance of the critical dispatcher loop, with only minimal additional per-quantum overhead. While we made no effort to support fine grained locking of the translation infrastructure, as only a small number of routines modify the translation cache state, the benefits of increased parallelism in this region compared to the simplicity of global locking are not compelling.

While the first naive approach of locking all store and atomic instructions preserves atomicity, performance suffers tremendously in any store-intensive application. Thread serialisation approaches offer better performance in tools observed thus far, and it is questionable whether performance will ever scale sufficiently to be of practical value.

Serialisation of atomics and insertion of host atomic instructions are shown to provide adequate safety and minimal overhead, allowing performance to scale better than native execution. As performance of both methods is quite close, the additional safety of host atomics outweighs the minimal amount of information potentially lost to instrumentation tools. Simultaneous memory manipulation by the kernel or other external agents is supported only in this atomicity, a problem which serial Valgrind currently ignores.

Performance analysis tools, such an instructions and cache profilers, can easily be extended for the parallel framework. Care must be taken however in data layout, with a separate data structure for each thread being required to avoid host false cacheline sharing overheads. While problems for shadow memory based tools remain, memory profiling tools such as Cachegrind have been shown to scale effectively for highly parallel applications.

Serial Valgrind's scheduling poses analysis problems for performance analysis tool. Over et. al. discuss accuracy tradeoffs with parallel simulators, noting a larger quantum tends to produce results which deviate from native execution [10]. This calls into question the accuracy of any analysis which depends on relative thread timing, including that of multiple cache behavior with shared data. While pValgrind does not provide timing guarantees, it allows the interleaving of thread memory effects and forward progress of live threads.

The source code for pValgrind, associated tools and support documentation will be made available from http://ccnuma.anu.edu.au/pvalgrind/.

## 9 Future Work

pValgrind will form the basis of further work evaluating scientific compute workloads. As with serial Valgrind, this can be used to directly analyse cache usage for an execution of a given application.

As pCachegrind cannot model data sharing we aim to extend our model to include cache coherency, to investigate inter-processor cache effects. This will capture events relating to cache sharing and performance for parallel application developers.

Coherency models could be further extended to drive models of NUMA platforms and interconnects, such as AMD Opteron systems. This will enable analysis of memory placement algorithms, topology selection and interconnect performance.

With the introduction of a timing model, parallel scheduling will be developed so that realistic thread timings may be simulated.

## 10 Acknowledgments

This work has been supported by ARC Linkage Grant LP0774896, in collaboration with Sun Microsystems. Many thanks to the Valgrind team for producing such a capable and productive framework.

## References

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[2] R. Bedichek. Simnow: Fast platform simulation purely in software. In *Hot Chips 2004*, 2004.

[3] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[4] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 265–275, March 2003.

[5] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[6] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[7] Markus Kowarschik Josef Weidendorfer and Carsten Trinitis. A tool suite for simulation based analysis of memory access behavior. In *Proceedings of the 4th International Conference on Computational Science (ICCS 2004)*, June 2004.

[8] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[9] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM Press.

[10] Andrew Over, Bill Clarke, and Peter E. Strazdins. A comparison of two approaches to parallel simulation of multiprocessors. In *ISPASS'07: International Symposium on Performance Analysis of Systems and Software*, pages 12–22, April 2007.

[11] Kevin Scott, Jack Davidson, and Kevin Skadron. Low-overhead software dynamic translation. Technical Report CS-2001-18, Department of Computer Science, University of Virginia, 2001.

[12] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[13] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996.