A Generalized Reduction Construct for Deterministic OpenMP

Amittai Aviram and Bryan Ford ({amittai.aviram,bryan.ford}@yale.edu)

January 28, 2012

Abstract

In parallel programming, a *reduction* is an operation that combines values across threads into a single result, and can be designed and implemented so as to enforce determinism, not only on the result, but also on the intermediate values and evaluation sequence. These features make the reduction an attractive feature for a language-based approach to deterministic parallelism, especially one that, like OpenMP, provides annotations to parallelize legacy serial code. Hence reductions are a planned feature of the Deterministic OpenMP (DOMP) project. To enable DOMP to help programmers eliminate nondeterministic code wherever possible, we propose a *generalized* reduction that supports arbitrary data types and user-defined operations—a generalization rich in challenges for language design and implementation.

1 Introduction

Deterministic OpenMP (DOMP) is a recently proposed approach to the problem of achieving efficient, flexible, user-friendly deterministic parallelism. It brings together previous developments from the two areas of programming languages and systems in order to offer a combination of convenience to the programmer and strict enforcement of race-freedom and determinism guarantees.

On the language side, DOMP is based on the well-known OpenMP specification [10, 22], a set of *annotations* and function calls that can be added to legacy serial code in languages such as C, C++, or Fortran, enabling the programmer to parallelize it conveniently and incrementally. OpenMP offers a range of constructs sufficiently broad to provide reasonable expressiveness in parallelizing code.

On the systems side, DOMP draws from the ideas of Workspace Consistency and working-copies determinism [4], a memory and programming model that eliminates race conditions by handling shared data, roughly speaking, as versioning systems handle shared documents—providing an isolated working copy for each thread, merging these copies at synchronization points, detecting any conflicting writes at merge time, and treating them as errors.

The OpenMP standard supports a *reduction* feature, which makes it possible to aggregate computed data across threads—for instance, to get the sum or product of individual values computed severally by concurrent threads. The reduction is an attractive feature for DOMP. Although it may look like a purposeful race condition, because each thread seems to update the reduction variable at the same time, it is in fact inherently deterministic: a sum or product will always be the same on every run with the same input, regardless of scheduling accidents, so long as the individual summands or factors are the same.

Reductions fit in well with DOMP's goals. But, in order to be compatible with working-copies determinism, DOMP's API must exclude certain other features of OpenMP because they are naturally nondeterministic—low-level race management constructs such as *atomic*, *critical*, and *flush*. Research suggests, however, that programmers often use these features only in order to implement, by hand, higher-level idioms that are, themselves, inherently deterministic, but for which OpenMP offers no ready-made constructs. One such case is a more general form of reduction. OpenMP only supports reductions for scalar value types, such as `int` and `double`, and for a restricted set of simple combining operations, such as addition (for sums) and multiplication (for products). If DOMP offered a generalized reduction for arbitrary types, including those accessible only by indirection, and for user-defined binary operations, then, we believe, programmers would be able to avoid many instances of unsafe, low-level synchronization abstractions.

A generalized, deterministic reduction construct comes with its own new questions and challenges, however, which the rest of this paper considers. Are there any constraints at all that DOMP must place upon the user-defined operations that a reduction can support? What should the API look like and what are possible tradeoffs in its semantics? What should

the fixed evaluation order be? How might it be implemented efficiently so as to take advantage of parallelism while safeguarding determinism? What sort of threading model is best suited for this purpose? What are the other major implementation challenges and how might we address them?

Section 2 reviews the background to these questions, and related work, in greater detail; Section 3 considers design-related challenges, such as the definition of the API; Section 4 does the same with implementation challenges; Section 5 reports on the project's current state; and Section 6 concludes.

2 Background

Research has established the desirability of achieving deterministic parallelism at a reasonable cost to performance [5, 11, 21]. Various teams have proposed distinct solution approaches, of which Deterministic OpenMP [3] is a recent one whose motivation we review here. We then take up the importance of reductions in general and within DOMP in particular.

2.1 Deterministic OpenMP

Among the earliest efforts to provide deterministic parallelism, language-based approaches are promising, but may apply only to special purposes, or may entail new and unfamiliar concepts, requiring extensive rewriting of legacy code [1, 2, 6, 12, 17, 19, 23, 24]. In order to support legacy code, other teams have developed deterministic replay systems, but these are generally more practical for development and debugging than for production systems [14, 16, 20, 25]. Deterministic scheduling systems also support legacy code [7, 8, 9, 15], but only allow race conditions to be reproduced (and perhaps silently executed) without eliminating them [3, 4, 5].

Working-copies determinism also supports legacy code in languages such as C, but it eliminates race conditions by catching them and treating them as runtime errors. Working-copies determinism is based on the Workspace Consistency memory model [4], and underlies the design of the Determinator operating system [5]. Workspace consistency requires that the programming language, implementation, and runtime system all remain *naturally deterministic*, meaning that interprocess or inter-thread synchronization events must occur in a manner that the program alone specifies, wholly immune to the scheduler or any hardware timing effects. Accordingly, a compatible program eschews such naturally nondeterministic synchronization abstractions as mutex locks and condition variables, relying instead on deterministic abstractions such as *fork*, *join*, and *barrier*. The execution environment then enforces race freedom by providing each concurrent thread with its own private logical copy of shared state at the *fork*, and merging the changes written by these threads into the emerging parent thread at *join* or *barrier*, checking for conflicting writes in the process and treating any such conflicts as errors. While the Determinator project demonstrated the efficient implementation of working-copies determinism, the programming model, constrained to this small set of synchronization abstractions, suffered from a lack of expressiveness and flexibility.

A deterministic version of OpenMP may remedy this shortcoming. The OpenMP standard defines a set of constructs, optional clauses, and routines that a programmer can add to a program written in a legacy language such as C, C++, or Fortran, in order to parallelize it easily and incrementally. The constructs annotate structured blocks. The fork-join synchronization pattern and the orientation toward structured blocks make for a convenient fit between working-copies determinism and OpenMP. For a deterministic version truly compatible with working-copies determinism, we must exclude the relatively few naturally nondeterministic constructs, such as *atomic*, *critical*, and *flush*, and provide an implementation that applies working-copies semantics to the remaining features. This is the Deterministic OpenMP (DOMP) project [3].

While DOMP promises greater expressiveness and ease of adaptation of legacy code for working-copies determinism, its exclusion of nondeterministic OpenMP features might seem to limit its usefulness in adapting real-world code. The research supporting DOMP, however, suggests that many instances of naturally nondeterministic synchronization abstractions actually arise from the need to implement higher-level idioms that are, themselves, naturally deterministic in principle, but for which the available language constructs do not offer direct support. With two such features, *generalized reductions* and *pipelines*, most uses of nondeterministic OpenMP constructs can be eliminated. The first of these is the concern of this paper.

2.2 Generalized Reductions

A *reduction*, also called a *fold* in functional programming, is a kind of higher-order operation whose special features make it a potentially useful language construct for deterministic parallelism. Given a data structure, such as a list z whose elements have type T

and a binary "combining" function $f : (T \times T) \to T$, a reduction $r : (f \times [T]) \to T$ applies f recursively to the elements of z, first to the first pair, and then to the result and the next element, etc., to arrive at the final result z_f:

$$z_f = r(z) = f\left(f\left(\ldots f\left(f\left(z_0, z_1\right), z_2\right), \ldots\right), z_{n-1}\right)$$

Parallel threads might compute the respective values of the list elements $z_0, \ldots, z_{n-1}$ concurrently, while r itself, expressing a series of dependencies, can be controlled by careful synchronization. Moreover, if the combining operation is both associative and commutative, the end result z_f remains the same regardless of the order of intermediate evaluation steps. Hence, so long as the intermediate results remain opaque and inaccessible, the entire reduction operation is effectively deterministic, even in an implementation that does nothing special to enforce or guarantee determinism.

This is the case with OpenMP, whose parallelizing constructs annotate otherwise serial structured blocks in a C, C++, or Fortran program. The optional `reduction` clause included in such an annotation specifies an operation and one or more variables, such that the compiled code will apply the reduction to the thread-local instances of that variable:

```
omp_set_num_threads(4);
int x = 0;
#pragma omp parallel reduction(+:x)
{ x += omp_get_thread_num() + 1; }
// x == 1 + 2 + 3 + 4 == 10
```

Semantically, `reduction` indicates that each thread in the ensuing block gets its own private copy of each reduction variable, initialized to the operation's identity value, such as 0 for +. At the end of the block, OpenMP combines all these private values using the specified operator, and combines the result with that variable's initial value from before the `parallel` block. OpenMP makes no guarantees about the evaluation order and thus treats intermediate values as undefined or indeterminate—but the final result will always be the same.

A truly deterministic parallel framework could, however, fix—and document—a predetermined evaluation order, so that intermediate states would be reproducible. Such a prescribed order could, itself, preserve some parallelism, e.g., combining values in a binary tree pattern, each rank of which executes in parallel. This would make sense for reduction semantics in DOMP.

OpenMP only supports reductions on scalar value types and a small set of simple, associative and commutative, arithmetic or logical operations: `+, -,*, &, |, &&, ||, ^`. (The operator `-` counts as commutative because the initial value for the reduction is 0; the initial value given in the program is *added* at the end.) These constraints make implementation fairly straightforward; but what if we want more for DOMP? For instance, vector addition, which is commutative and associative, seems but a small step away; but, without support for pointers and aggregate types, we must implement the equivalent using low-level, *naturally nondeterministic* and therefore error-prone, synchronization abstractions, such as the `atomic` or `critical` construct. Furthermore, if we follow a fixed evaluation order and make it known to the programmer, we can even relax the constraints on operations, and thus extend the usefulness of reductions. For instance, matrix multiplication is associative but not commutative. If the evaluation order is not only known but reasonably intuitive with respect to the written program, the programmer can put the factor matrices in the right places and enjoy the benefits of deterministic parallelism. Such a design follows the "principle of least surprise": although the evaluation order is fixed independently of the operation and data, the programmer can work with it in mind so as to get the correct results—reliably and reproducibly.

3 Design Challenges

Defining an appropriate reduction feature for DOMP raises several issues in language design and its implications for possible implementation. In particular, we consider (a) the constraints on allowable combination functions, (b) the choice of a suitably general API and (c) possible evaluation orders.

3.1 Constraints on Functions

In saying that DOMP reductions should support "arbitrary" user-defined combination functions, we assume some unspoken constraints. For instance, the definition of reduction (2.2) stipulates that the function's two arguments and return value all have the same type. In addition, the OpenMP specification's description of the semantics of reductions requires that the combination function have an identity element of the same type. This is necessary for OpenMP's way of avoiding the duplication of the initial value to work: each thread's private copy of the reduction variable must be initialized to the identity value, and the original initial value must be brought into the computation at the end. There are other ways to avoid duplication—for instance, by applying the *inverse*of the combination function whenever the

master joins any other thread in the team [13], an approach which generalizes beyond OpenMP's team threading model to arbitrary forking and joining patterns. In that case, however, every function must have an inverse, and it is generally easier to find an identity element than an inverse function. Multiplication of square matrices has a known identity element, but an inverse may not exist; if it does, finding it may not be trivial.

As noted, OpenMP requires that the combination function be both commutative and associative, but we propose relaxing this constraint to allow functions that are not commutative, such as matrix multiplication, so long as the programmer knows the evaluation order in advance. Must the function be associative? If not, we might get incorrect results when parallelizing intermediate evaluation steps. Fortunately, this constraint is not hard to satisfy in most ordinary programs.

3.2 API

Since OpenMP and DOMP work with C-style languages, supporting arbitrary and not just scalar value types means having to deal with pointers. To keep the API simple, then, we propose that the combination function have the most general of signatures:

```
void f(void * cum, const void * new);
```

A call $f(c, n)$ changes the value of c as a side effect to reflect the application of f on c and n. While this specification accommodates composite types and keeps the rules straightforward, it means that simple scalars would have to be passed effectively by reference rather than by value, slightly complicating the code.

Since the user may define any combination function within these constraints, he or she must supply the appropriate identity element, as well as the reduction variable. Moreover, DOMP's working-copies determinism model requires that the runtime make thread-local copies of the identity element, and, for each thread, re-point the reduction variable to point to this local, or *scratch*, copy. (This is equivalent to OpenMP's initializing thread-local copies of the reduction variable to the identity value.) The scratch object serves as an accumulator in the merge process. At the end of the evaluation sequence, the master passes the reduction variable's original object and the scratch object, containing the penultimate cumulative result, to the combination function; stores the final result in the original object's location; and repoints the reduction variable again to point there. And, since the master must make these local scratch copies of an *arbitrary* identity object, it is most practical to store the identity object in contiguous memory, and inform the master of the object's size.

The reduction call itself, then, takes the form

```
omp_reduction(void (*f)(void *, void*),
     void ** var, void ** identity,
     size_t size);
```

The master saves the reduction variable pointer's address, the address of the original reduction variable object, the scratch object's address, and f in a `reduction_var` data structure for use during evaluation.

3.3 Evaluation Order

A good fixed evaluation order should try to satisfy two different demands: (a) intuitive clarity for the programmer, and (b) opportunities to parallelize the evaluation as much as possible. For the first, it makes sense to have the order of evaluation follow the order of thread IDs, provided that the programmer can know or discover this. For instance, given a number of `setion`s within a `sections` construct less than or equal to the number of available threads, DOMP could guarantee that the first `section` went to thread 0 (the master), the second to thread 1, etc., and that the runtime's evaluation order would be *equivalent* to that of the sequential execution of the `section`s by a single thread.

The simplest way of going about this is to have the master, at the end of a parallel block, aggregate the result sequentially. Using the symbol $\square$ to represent the combination function, we then would have

$$z_f = (\ldots((z_0 \square z_1) \square z_2) \ldots \square z_{n-1})$$

In fact, GCC's *libgomp* implementation serializes in this manner, but without any fixed order at all, having each thread use low-level atomic instructions to update the original reduction variable after it has finished computing on its private copy. But this approach, even if made deterministic, foregoes parallelism entirely. Using the function's associativity, we can equivalently evaluate in pairs, and then again in pairs of results, etc., following a binary tree. For 8 threads, for instance,

$$z_f = (((z_0 \square z_1) \square (z_2 \square z_3)) \square ((z_4 \square z_5) \square (z_6 \square z_7)))$$

Note that this evaluation order does not move the elements from their original order and therefore does not imply commutativity.

As it happens, this binary tree coincides nicely with an efficient implementation of the merge phase in

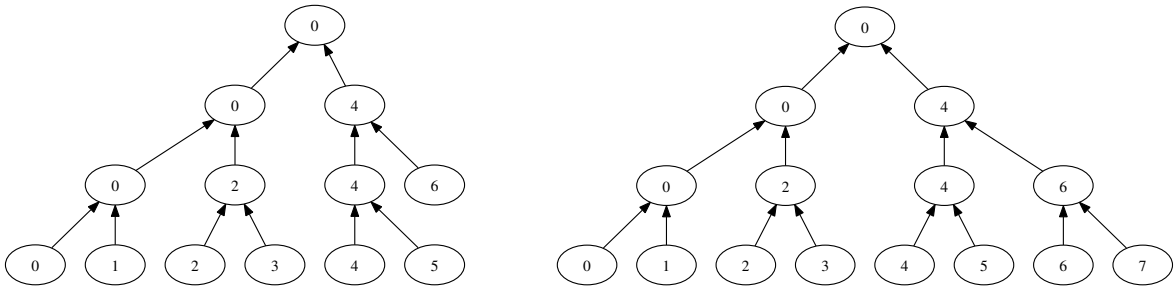

Figure 1: DOMP merge scheme for (a) 7 and (b) 8 threads. Efficiently parallel evaluation of reductions can coincide with merging.

working-copies determinism. In this scheme, every even-numbered thread merges another thread into itself at least once. Threads whose IDs are higher powers of two repeat this procedure, and the master, whose ID is 0, merges into itself, in order, every thread whose ID is a power of two. Figure 1 represents this scheme for 7 and 8 threads, respectively. Every rank of the tree corresponds to parallel execution. For reductions, each merge operation includes an application of the combination function f for the two versions of reduction variable v:

$$f(v_{\text{self}}, v_{\text{other}})$$

At the end, the master applies

$$f(v_{\text{init}}, v_{\text{self}})$$

This scheme preserves the program's expressed order.

4 Implementation Challenges

The DOMP project in general, and the DOMP reduction in particular, present a number of challenges for implementation. Whereas conventional OpenMP implementations such as GCC's *libgomp* use an underlying shared-process threading model based on pthreads, this may not be ideally suited to our current needs. Whether the implementation uses lightweight threads or processes, moreover, the performance may benefit from maintenance of a *thread pool* even after the first parallel block is finished, for use in later parallel blocks. Finally, at the end of each parallel block, at merge time, the thread merging another thread's data into its own (while checking for conflicting writes) would ordinarily always spot a conflict on the reduction variable, unless the latter is given special treatment. We now consider each of these issues.

4.1 Threads or Processes

Without working-copies determinism, and if the allowable combining functions and their respective identity values are narrowly constrained as they are in standard OpenMP, it is practical for the compiler to create thread-local "copies" of each reduction variable by simply allocating space in each thread's stack and initializing it to the known default value. One of the compiler's transformation steps could turn this code

```
int x = 42;
#pragma omp parallel
{ x += 3; }
```

into something like this:

```
struct data_t { int x; } data;
int x = 42;
data.x = x;
for (int i = 0; i < num_threads; ++i)
  pthread_create(&threads[i], &attr,
    &f0, &data);
/* ... */
void * f0(void * data) {
  int x = 0;
  x += 3;
  atomic_op(data->x += x); }
```

But for working-copies determinism, each of DOMP's threads needs to have thread-private copies of *all shared state*, including objects to which pointer variables refer. Since the latter makes object-by-object copying impossible, the compiler or runtime would have to copy entire memory regions (data segments) for each thread, along with thread-specific address translations. A runtime virtual machine might do the address translation with reasonable efficiency [18]. Alternatively, and perhaps more simply, the implementation could spawn new processes using Unix `fork`, which would allow each thread to inherit its

own private copy of shared state automatically. This approach shifts the complexity away from copying and toward interprocess communication and synchronization at merge time—and afterwards, to broadcast updates to shared state, if we maintain a thread pool for the next parallel block (discussed below).

4.2 Thread Pools

Once the master process has spawned enough child processes to make a team, it seems clearly advantageous to keep the team around after the end of a parallel block, in order to avoid spawning it anew on a later parallel block. The allocation and initialization of reduction variable and identity objects can occur in any serial execution context, including one between two parallel blocks. In this case, the master must update processes waiting in the thread pool with the changed state before they restart parallel execution. These updates include, not only the obects themselves, but also the values of the variables pointing to them. We can accomplish this with reasonable efficiency by copying the relevant virtual memory pages from the master's address space to a *scratch file*, and then having the team threads copy those pages from the scratch file to their own address spaces, asynchronously and in parallel, before resuming execution. DOMP uses the same mechanism to update waiting pooled threads about dynamic memory allocation or other state changes between parallel blocks. A common way of catching all such changes in state is to write-protect the master's data segments between parallel blocks, trap on each write, and record the page of each trap. Since the relevant portion of the stack is usually small (spanning two pages at most), the master can include the stack in updates by default, thus avoiding at least some traps.

4.3 Merging around the Reduction

At the heart of the merge process in DOMP's implementation of working-copies determinism is a loop that goes over each byte (or other-size chunk, depending on the merge granularity) of the thread's own data (*self*), the other thread's corresponding data (*other*), and a pristine reference copy (*ref*), tucked away at the start of the parallel block and representing that earlier state of the data:

```
for (; self != end; ++self, ++other,
    ++ref) {
  if (*other != *ref) {
    if (*self != *ref)
      race_exception();
    else *self = *buddy;
  }
}
```

When the loop reaches the address of the identity object to which the reduction variable points, it will surely detect a race condition—not what we intend!

To avoid this problem, each merging thread resolves all reductions with its paired thread before entering the merge loop. Then, when the merge loop detects a conflict, it first checks the conflicting address against the list of scratch objects. If a conflicting address falls within the range of a scratch object, the merge loop skips forward to that object's end. With the globally-visible list of `reduction_var` structures sorted on scratch object addresses, the merging thread can evaluate, check, and skip with reasonable efficiency.

5 Current State

We currently have a complete implementation of the core features of DOMP, including generalized reductions, and have successfully tested cumulative matrix multiplication. To implement DOMP, we leveraged GCC's parsing mechanism by altering only its OpenMP support library `libgomp`, replacing relevant Gnu implementations with our own DOMP implementations of `libgomp`'s internal API. We plan soon to demonstrate and evaluate the replacement of unsafe low-level synchronization constructs with DOMP reductions in NPB benchmarks.

6 Conclusion

A generalized reduction is an attractive feature for Deterministic OpenMP, one that furthers its goal of combined convenience and strict, race-free deterministic parallelism. Although design and implementation challenges are not trivial, we have seen possible ways to overcome these, so as to help fulfill the promise of DOMP and working-copies determinism.

References

[1] W.B. Ackerman. Data flow languages. *Computer*, 15(2):15–25, Feb 1982.

[2] Zachary Anderson, David Gay, Rob Ennals, and Eric Brewer. SharC: checking data sharing strategies for multithreaded C. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 149–158, New York, NY, USA, 2008. ACM.

[3] Amittai Aviram and Bryan Ford. Deterministic OpenMP for race-free parallelism. In *3rd HotPar*, May 2011.

[4] Amittai Aviram, Bryan Ford, and Yu Zhang. Workspace Consistency: A programming model for shared memory parallelism. In *2nd WoDet*, March 2011.

[5] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Determinator: OS support for efficient deterministic parallelism. In *9th OSDI*, October 2010.

[6] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of Java without data races. In *OOPSLA '00*, pages 382–400, 2000.

[7] Tom Bergan et al. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *15th ASPLOS*, March 2010.

[8] Tom Bergan et al. Deterministic process groups in dOS. In *9th OSDI*, October 2010.

[9] Emery D. Berger et al. Grace: Safe multithreaded programming for C/C++. In *OOPSLA*, October 2009.

[10] OpenMP Architecture Review Board. OpenMP application program interface version 2.5, May 2005. `http://www.openmp.org/mp-documents/spec25.pdf`.

[11] Robert L. Bocchino et al. Parallel programming must be deterministic by default. In *HotPar*. USENIX, March 2009.

[12] Robert L. Bocchino et al. A type and effect system for deterministic parallel Java. In *OOPSLA*, October 2009.

[13] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In *OOPSLA 2010*, pages 691–707, 2010.

[14] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of Java multithreaded applications. In *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 48–59, New York, NY, USA, 1998. ACM.

[15] Joseph Devietti et al. DMP: Deterministic shared memory multiprocessing. In *14th ASPLOS*, March 2009.

[16] George W. Dunlap et al. Execution replay for multiprocessor virtual machines. In *VEE*, March 2008.

[17] Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. *Transactions on VLSI Systems*, 14(8):854–867, August 2006.

[18] Bryan Ford and Russ Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX*, June 2008.

[19] P. Hudak. Para-functional programming. *Computer*, 19(8):60–70, August 1986.

[20] Thomas J. Leblanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[21] E.A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[22] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[23] Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, 1998.

[24] Paul Roe. *Parallel Programming using Functional Languages*. PhD thesis, University of Glasgow, feb 1991. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.46.7763&rep=rep1&type=pdf`.

[25] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA '03*, pages 122–135, June 2003.