Eliminating Synchronization Overhead in Automatically Parallelized Programs Using Dynamic Feedback

PEDRO C. DINIZ
Information Sciences Institute
University of Southern California
and
MARTIN C. RINARD
Laboratory for Computer Science
Massachusetts Institute of Technology

This article presents dynamic feedback, a technique that enables computations to adapt dynamically to different execution environments. A compiler that uses dynamic feedback produces several different versions of the same source code; each version uses a different optimization policy. The generated code alternately performs sampling phases and production phases. Each sampling phase measures the overhead of each version in the current environment. Each production phase uses the version with the least overhead in the previous sampling phase. The computation periodically resamples to adjust dynamically to changes in the environment.

We have implemented dynamic feedback in the context of a parallelizing compiler for object-based programs. The generated code uses dynamic feedback to automatically choose the best synchronization optimization policy. Our experimental results show that the synchronization optimization policy has a significant impact on the overall performance of the computation, that the best policy varies from program to program, that the compiler is unable to statically choose the best policy, and that dynamic feedback enables the generated code to exhibit performance that is comparable to that of code that has been manually tuned to use the best policy. We have also performed a theoretical analysis which provides, under certain assumptions, a guaranteed optimality bound for dynamic feedback relative to a hypothetical (and unrealizable) optimal algorithm that uses the best policy at every point during the execution.

Categories and Subject Descriptors: D.3.4 [**Compilers**]: Parallelizing Compilers

General Terms: Parallelizing compilers, compilers, parallel computing

1. INTRODUCTION

The most efficient implementation of a given abstraction often depends on the environment in which it is used. For example, the best consistency protocol in a software distributed shared memory system often depends on the access pattern of the parallel program [Falsafi et al. 1994]. The best data distribution of dense matrices in distributed memory machines depends on how the different parts of the program access the matrices [Amarasinghe and Lam 1993; Anderson and Lam 1993; Gupta and Banerjee 1992; Kennedy and Kremer 1995]. The best concrete data structure to implement a given abstract data type often depends on how it is used [Freudenberger et al. 1983; Kiczales 1986]. The best algorithm to solve a given problem often depends on the combination of input and hardware platform used to execute the algorithm [Brewer 1995]. In all of these cases, it is impossible to statically choose the best implementation — the best implementation depends on information (such as the input data, dynamic program characteristics, or hardware features) that is either difficult to extract or unavailable at compile time. If a programmer has a program with these characteristics, he or she is currently faced with two unattractive alternatives: either

manually tune the program for each environment, or settle for suboptimal performance.

We have developed a technique, *dynamic feedback*, that enables programs to automatically adapt to different execution environments. A compiler that uses dynamic feedback produces several different versions of the same code. Each version uses a different optimization policy. The generated code alternately performs *sampling* phases and *production* phases. During a sampling phase, the generated code measures the overhead of each version in the current environment by running that version for a fixed time interval. Each production phase then uses the version with the least overhead in the previous sampling phase. After running a production phase for a fixed time interval, the generated code performs another sampling phase. If the environment has changed, the generated code dynamically adapts by using a different version in the next production phase.

We see dynamic feedback as part of a general trend towards adaptive computing. As the complexity of systems and the capabilities of compilers increase, compiler developers will find that they can automatically apply a large range of transformations, but have no good way of statically determining which transformations will deliver good results when the program is actually executed. The problem will become even more acute with the emergence of new computing paradigms such as mobile programs in the Internet. Mobile programs will be expected to execute efficiently on a wide range of hardware platforms and execution environments. The extreme heterogeneity of these systems will make it difficult, if not impossible, to generate optimal code for all platforms. Dynamic feedback is one example of the adaptive techniques that will enable the generated code to deliver good performance in a variety of different environments.

In this article we illustrate the application of dynamic feedback to a particular set of program transformations: synchronization transformations for parallel programs. These transformations occur in the context of a parallelizing compiler for object-based languages. The compiler generates parallel code that uses synchronization constructs to make operations execute atomically [Rinard and Diniz 1997]. Our experimental results show that the resulting synchronization overhead can significantly degrade the performance [Diniz and Rinard 1998; 1997]. We have developed a set of synchronization transformations and a set of synchronization optimization policies that use the transformations to reduce the synchronization overhead [Diniz and Rinard 1998; 1997].

Unfortunately, the best policy is different for different programs, and may even vary dynamically for different parts of the same program. Furthermore, the best policy depends on information, such as the global topology of the manipulated data structures and the dynamic execution schedule of the parallel tasks, that is unavailable at compile time. The compiler is therefore unable to statically choose the best synchronization optimization policy.

Our implemented compiler generates code that uses dynamic feedback to automatically choose the best synchronization optimization policy. Our experimental results show that dynamic feedback enables the automatically generated code to exhibit performance comparable to that of code that has been manually tuned to use the best policy.

1.1 Contributions

This article makes the following contributions:

—It presents a technique, dynamic feedback, that enables systems to automatically evaluate several different implementations of the same source code, then use the evaluation

to choose the best implementation for the current environment.

—It shows how to apply dynamic feedback in the context of a parallelizing compiler for object-based programs. The generated code uses dynamic feedback to automatically choose the best synchronization optimization policy.

—It presents a theoretical analysis that characterizes the worst-case performance of systems that use dynamic feedback. This analysis provides, under certain assumptions, a guaranteed optimality bound for dynamic feedback relative to a hypothetical (and unrealizable) optimal algorithm that uses the best policy at every point during the execution.

—It presents experimental results for the automatically generated parallel code. These results show that, for our set of benchmark applications,

  —it is possible to implement dynamic feedback with very low overhead, and

  —the performance of the code that uses dynamic feeedback is comparable to the performance of code that has been manually tuned to use the best synchronization optimization policy.

1.2 Scope

This article presents an application of dynamic feedback to a specific problem, the choice of the synchronization optimization policy, and to a specific class of programs, irregular scientific applications with parallel loops that contain synchronized atomic operations. Characteristics such as a small number of possible synchronization optimization policies, the use of parallel loops as a primary control structure, the lock acquisition frequency, the existence of a performance metric that allows the system to compare the performance of different policies even when the policies are used for different computations, and the fact that there is usually a single best optimization policy for each parallel loop, make dynamic feedback work well for this combination of optimization problem and application class. We expect that there may be significant issues that must be addressed when generalizing the specific techniques presented in this article to other optimization problems and other application and system contexts. We postpone a more detailed discussion of these issues until we have presented the important details of our specific approach. Section 4.6 presents this discussion.

1.3 Structure

The remainder of the article is structured as follows. In Section 2 we briefly summarize the analysis technique, commutativity analysis, that our compiler is based on. Section 3 presents the issues associated with synchronization optimizations and describes the different synchronization optimization policies. Section 4 discusses the basic issues that affect the performance impact of dynamic feedback. Section 5 presents a theoretical analysis that provides, under certain assumptions, a performance bound for dynamic feedback relative to the optimal algorithm, which uses the best policy at every point in the computation. Section 6 presents the experimental results. We discuss related work in Section 7 and conclude in Section 8.

2. COMMUTATIVITY ANALYSIS

We next provide an overview of *commutativity analysis*, the technique that our compiler uses to automatically parallelize our set of applications [Rinard and Diniz 1997]. This technique is designed to parallelize object-based programs. It analyzes the program at

the granularity of *operations* on *objects* to determine if the operations commute, i.e., if they generate the same result regardless of the order in which the operations commute. If all operations in a given computation commute, the compiler can automatically generate parallel code.

To test that two operations A and B commute, the compiler must consider two execution orders: the execution order A;B in which A executes first, then B executes, and the execution order B;A in which B executes first, then A executes. The two operations commute if they meet the following commutativity testing conditions:

—**Instance Variables:** The new value of each instance variable of the receiver objects of A and B under the execution order A;B must be the same as the new value under the execution order B;A.

—**Invoked Operations:** The multiset of operations directly invoked by either A or B under the execution order A;B must be the same as the multiset of operations directly invoked by either A or B under the execution order B;A.

Both commutativity testing conditions are trivially satisfied if the two operations have different receiver objects or if neither operation writes an instance variable that the other accesses — in both of these cases the operations are independent. If the operations may not be independent, the compiler reasons about the values computed in the two execution orders.

The compiler uses symbolic execution [Kemmerer and Eckmann 1985] to extract expressions that denote the new values of instance variables and the multiset of invoked operations. Symbolic execution simply executes the methods, computing with expressions instead of values. It maintains a set of bindings that map variables to expressions that denote their values and updates the bindings as it executes the methods. The compiler uses the extracted expressions from the symbolic execution to apply the commutativity testing conditions presented above. If the compiler cannot determine that corresponding expressions are equivalent, it must conservatively assume that the two operations do not commute.

The compiler uses commutativity analysis as outlined above to parallelize regions of the program. For each operation invocation site, it traverses the call graph to compute the set of operations in the computation rooted at that site. It then uses commutativity analysis to determine if all pairs of operations commute. If so, the compiler can generate code that executes all of the operations in that computation in parallel.

To ensure that operations in parallel computations execute atomically, the compiler augments each object with a *mutual exclusion lock*. It then automatically inserts synchronization constructs into operations that update objects. These operations first acquire the object's lock, perform the update, then release the lock. The synchronization constructs ensure that the operation executes atomically with respect to all other operations that access the object. Figure 1 presents a general example of the generated parallel code.

The compiler also applies an optimization that exposes parallel loops to the run-time system. If a `for` loop contains nothing but invocations of parallel versions of methods, the compiler generates parallel loop code instead of code that serially spawns the invoked operations. The run-time system can then apply standard loop scheduling techniques such as guided self-scheduling [Polychronopoulos and Kuck 1987]. A barrier at the end of the loop ensures that all of the iterations complete before the computation continues its execution after the loop.

```
class Object {
  private:
    iv_1, ..., iv_n; // instance variables
  public:
    void Method(p_1, ..., p_k);
};

void Object::Method(p_1, ..., p_k){
  // receiver object update
  l_i = expr(iv_1, ..., iv_k, p_1, ..., p_k);
  iv_i = expr(iv_1, ..., iv_n, p_1, ..., p_k);
  // operation invocations
  recv_i->Method_i(expr_i^1, ..., expr_i^{k_i});
    ⋮
  recv_j->Method_j(expr_j^1, ..., expr_j^{k_j});
}
```

```
class lock {
  public:
    void acquire();
    void release();
};

class Object {
  private:
    lock mutex; // mutual exclusion lock
    iv_1, ..., iv_n; // instance variables
  public:
    void ParMethod(p_1, ..., p_k);
};

void Object::ParMethod(p_1, ..., p_k){
  // receiver object update, executed atomically
  mutex.acquire();
  l_i = expr(iv_1, ..., iv_k, p_1, ..., p_k);
  iv_i = expr(iv_1, ..., iv_n, p_1, ..., p_k);
  mutex.release();
  // operation invocations, spawned concurrently
  spawn(recv_i->Method_i(expr_i^1, ..., expr_i^{k_i});
    ⋮
  spawn(recv_j->Method_j(expr_j^1, ..., expr_j^{k_j});
}
```

Fig. 1. Generic Parallel Code (parallel constructs in **bold** font)

In our applications, parallel loops are the primary source of available concurrency. The compiler therefore exploits concurrency only within parallel loops. The generated code executes a sequence of parallel and serial sections. Parallel sections correspond to the execution of parallel loops; serial sections correspond to the execution of serial code between parallel loops.

3. SYNCHRONIZATION OPTIMIZATIONS

We found that, in practice, the overhead generated by the synchronization constructs often reduced the performance. We therefore developed several synchronization optimization algorithms [Diniz and Rinard 1998; 1997]. These algorithms are designed for parallel programs, such as those generated by our compiler, that use mutual exclusion locks to implement *critical regions*. Each critical region acquires its mutual exclusion lock, performs its computation, then releases the lock.

Computations that use mutual exclusion locks may incur two kinds of overhead: *locking* overhead and *waiting* overhead. Locking overhead is the overhead generated by the execution of constructs that successfully acquire or release a lock. Waiting overhead is the overhead generated when one processor waits to acquire a lock held by another processor.

If a computation releases a lock, then reacquires the same lock, it is possible to reduce the locking overhead by eliminating the release and acquire. Our synchronization optimization algorithms statically detect computations that repeatedly release and reacquire

the same lock. They then apply *lock elimination transformations* to eliminate the intermediate release and acquire constructs [Diniz and Rinard 1998]. The goal of the lock elimination algorithm is to reduce the number of times the computation releases and acquires locks. The basic idea is to identify a computation that contains multiple critical regions that acquire and release the same lock, then transform the computation so that it contains one large critical section that acquires and releases the lock only once. Because the transformed computation acquires and releases the lock fewer times, it generates less lock overhead. Given a region over which to eliminate synchronization constructs, the algorithm uses the lock movement transformation to increase the sizes of critical regions that acquire and release the same lock until they are adjacent in the Interprocedural Control-Flow Graph (ICFG). [1] It then uses the lock cancellation transformation to eliminate adjacent release and acquire nodes. In effect, this optimization coalesces multiple critical regions that acquire and release the same lock multiple times into a single larger critical region that includes all of the original critical regions. The larger critical region, of course, acquires and releases the lock only once. This reduction in the number of times that the computation acquires and releases locks translates directly into a reduction in the locking overhead.

Figures 2 and 3 present an example of how synchronization optimizations can reduce the number of executed acquire and release constructs. Figure 2 presents a program (inspired by the Barnes-Hut application described in Section 6) that uses mutual exclusion locks to make `body::one_interaction` operations execute atomically. Figure 3 presents the program after the application of a synchronization optimization algorithm. The optimization algorithm interprocedurally lifts the acquire and release constructs out of the loop in the `body::interactions` operation. This transformation reduces the number of times that the `acquire` and `release` constructs are executed.

An overly aggressive synchronization optimization algorithm may introduce *false exclusion*. False exclusion may occur when a processor holds a lock during an extended period of computation that was originally part of no critical region. If another processor attempts to execute a critical region that uses the same lock, it must wait for the first processor to release the lock even though the first processor is not executing a computation that needs to be in a critical region. The result is an increase in the waiting overhead. Excessive false exclusion reduces the amount of available concurrency, which can in turn decrease the overall performance.

The synchronization optimization algorithms must therefore mediate a trade-off between the locking overhead and the waiting overhead. Transformations that reduce the locking overhead may increase the waiting overhead, and vice-versa. The synchronization optimization algorithms differ in the policies that govern their use of the lock elimination transformation:

—**Original:** Never apply the transformation — always use the default placement of ac-

[1] The ICFG is a generalization of the standard control-flow graph to include control-flow edges from call sites to the entry nodes of invoked operations, and from exit nodes of invoked operations back to the nodes after the call sites. The compiler constructs the ICFG in two phases. During the first phase, the compiler constructs the control-flow graph for each operation. It splits each operation invocation statement into two separate nodes: a call node and a return node. In the second phase, the compiler inserts an edge from the call node to the entry node of the invoked operation, and an edge from the exit node of the invoked operation back to the return node in the caller.

```
extern double interact(double,double);
class body {
  private:
    lock mutex;
    double pos,sum;
  public:
    void one_interaction(body *b);
    void interactions(body b[], int n);
};

void body::one_interaction(body *b) {
  double val = interact(this->pos, b->pos);
  mutex.acquire();
  sum = sum + val;
  mutex.release();
}

void body::interactions(body b[], int n) {
  for (int i = 0; i < n; i++) {
    this->one_interaction(&b[i]);
  }
}
```

Fig. 2. Unoptimized Example Computation

```
extern double interact(double,double);
class body {
  private:
    lock mutex;
    double pos,sum;
  public:
    void one_interaction(body *b);
    void interactions(body b[], int n);
};

void body::one_interaction(body *b) {
  double val = interact(this->pos, b->pos);
  sum = sum + val;
}

void body::interactions(body b[], int n) {
  mutex.acquire();
  for (int i = 0; i < n; i++) {
    this->one_interaction(&b[i]);
  }
  mutex.release();
}
```

Fig. 3. Optimized Example Computation

quire and release constructs. In the default placement, each operation that updates an object acquires and releases that object's lock.

—**Bounded:** Apply the transformation only if the new critical region will contain no cycles in the call graph. The idea is to limit the severity of any false exclusion by limiting the dynamic size of the critical region.

—**Aggressive:** Always apply the transformation.

In general, the amount of overhead depends on complicated dynamic properties of the computation such as the global topology of the manipulated data structures and the run-time scheduling of the parallel tasks. Our experimental results show that the synchronization optimizations have a large impact on the performance of our benchmark applications. Unfortunately, there is no one best policy. Because the best policy depends on information that is not available at compile time, the compiler is unable to statically choose the best policy.

4. IMPLEMENTING DYNAMIC FEEDBACK

The compiler generates code that executes an alternating sequence of serial and parallel sections. Within each parallel section, the generated code uses dynamic feedback to automatically choose the best synchronization optimization policy. The execution starts with a sampling phase, then continues with a production phase. During the sampling phase, code compiled with each policy executes for a fixed time interval. The parallel section measures the overhead of each policy, and uses the best policy during the production phase. The parallel section periodically resamples to adapt to changes in the best policy. We next discuss the specific issues associated with implementing this general approach.

4.1 Detecting Interval Expiration

The generated code for the sampling phase executes each policy for a fixed sampling time interval. The production phase also executes for a fixed production time interval, although the production intervals are typically much longer than the sampling intervals. The compiler uses two values to control the lengths of the sampling and production intervals: the *target sampling interval* and the *target production interval*. At the start of each interval, the generated code reads a timer to obtain the starting time. As it executes, the code periodically *polls* the timer: it reads the timer, computes the difference of the current time and the starting time, then compares the difference with the target interval. The comparison enables the code to detect when the interval has expired. Several implementation issues determine the effectiveness of this approach:

—**Potential Switch Points:** In general, it is possible to switch policies only at specific *potential switch points* during the execution of the program. The rate at which the potential switch points occur in the execution determines the minimum polling rate, which in turn determines how quickly the generated code responds to the expiration of the current interval.

In all of our benchmark applications, each parallel section executes a parallel loop. A potential switch point occurs at each iteration of the loop, and the generated code tests for the expiration of the current interval each time it completes an iteration. A potential drawback of using polling to determine interval expiration is that if the execution of a single iteration is significantly longer than the target sampling or production interval,

the code will be able to react only after it has successfully completed the computation alloted to the sampled code.[2] In our benchmark applications, the individual iterations of the loops are small enough so that each processor can respond reasonably quickly to the expiration of the interval.

Code that uses parallel operations instead of parallel loops requires a more evolved code generation scheme. For the problem of choosing the synchronization optimization policy, a potential switch point would occur at the beginning of each parallel operation.[3] The compiler would therefore generate code that checks for the expiration of the current interval at the start of each parallel operation instead of at the start of each loop iteration.

—**Polling Overhead:** The polling overhead is determined in large part by the overhead of reading the timer. Our currently generated code uses the timer on the Stanford DASH machine. The overhead of accessing this timer is approximately nine microseconds, which, as we report in Section 6, is negligible compared with the sizes of the iterations of the parallel loops in our benchmark set of applications.

—**Synchronous Switching:** The generated code switches policies synchronously. When an interval expires, each processor waits at a barrier until all of the other processors detect that the interval has expired and arrive at the barrier. This strategy ensures that all processors use the same policy during each sampling interval. The measured overhead therefore accurately reflects the overhead of the policy. Synchronous switching also avoids the possibility of interference between incompatible policies.[4]

Synchronous switching poses two potential drawbacks – barrier waiting and barrier overhead. Barrier waiting occurs when each processor must wait for all of the other processors to detect the expiration of the current interval before it can proceed to the next interval. This effect can have a significant negative impact on the performance if one of the iterations of the parallel loop executes for a long time relative to the other iterations and to the sampling interval. The combination of an especially bad policy (for example, a synchronization optimization policy that serializes the computation) and iterations of the loop that execute for a significant time relative to the sampling interval can also cause poor performance.

Barrier overhead is the inherent overhead of the barrier construct itself. On the Stanford DASH machine, we implemented the barrier by maintaining a count of the number of processors that reach the barrier. We used a lock to make the count operations atomic. For this implementation, the overhead of the barrier is negligible compared to the other overheads associated with the use of dynamic feedback.

—**Timer Precision:** The precision of the timer places a lower bound on the size of each interval. The timer must tick at least once before the interval expires. In general, we do not expect the precision of the timer to cause any problems. Our generated code uses

[2] An interrupt driven implementation for computations that acquire and release mutual exclusion locks presents itself as a challenging task. Complications include the possibility of a timer interrupt occurring in the middle of a critical region or at a state from which a potential switch point can be reached only after unwinding many stack frames from the call stack.

[3] We assume here that the compiler would serialize all computation that takes place inside a critical region.

[4] This potential problem does not arise when using dynamic feedback to choose the best synchronization optimization policy. All of the synchronization optimization policies are compatible: it is possible to concurrently execute different versions without affecting the correctness of the computation. We expect that in other applications of dynamic feedback, however, the different policies may be incompatible and the concurrent execution of different versions may cause the computation to execute incorrectly.

target sampling intervals of at least several milliseconds in length. Most systems provide timers with at least this resolution.

All of these issues combine to determine the *effective sampling interval*, or the minimum time from the start of the interval to the time when all of the processors detect that the interval has expired and proceed to the next interval.

4.2 Switching Policies

During the sampling phase, the generated code must switch quickly between different synchronization optimization policies. The current compiler generates three versions of each parallel section of code. Each version uses a different synchronization optimization policy.

The advantage of this approach is that the code for each policy is always available, which enables the compiler to switch very quickly between different policies. The currently generated code simply executes a **switch** statement at each parallel loop iteration to dispatch to the code that implements the current policy.

The potential disadvantage is an increase in the size of the generated code. Table I presents the sizes of the text segments for several different versions of the benchmark applications from Section 6. These data are from the object files of the compiled applications before linking and therefore include code only from the applications — there is no code from libraries. The Serial version is the original serial program, the Original version uses the Original synchronization optimization policy, and the Dynamic version uses dynamic feedback. In general, the increases in the code size are quite small. This is due, in part, to an algorithm in the compiler that locates closed subgraphs of the call graph that are the same for all optimization policies. The compiler generates a single version of each method in the subgraph, instead of one version per synchronization optimization policy.

Application	Version	Size (bytes)
Barnes-Hut	Serial	25,248
	Original	31,152
	Dynamic	33,648
Water	Serial	36,832
	Original	46,960
	Dynamic	50,784
String	Serial	36,064
	Original	43,616
	Dynamic	45,664

Table I. Executable Code Sizes (bytes)

We also considered using dynamic compilation [Auslander et al. 1996; Engler 1996; Lee and Leone 1996] to produce the different versions of the parallel sections as they were required. Although this approach would reduce the amount of code present at any given point in time, it would significantly increase the amount of time required to switch policies in the sampling phases. This alternative would therefore become viable only for situations in which the minimum sampling phases could be significantly longer than we wished.

Finally, it is possible for the compiler to generate a single parameterized version of the code that can use any of the three synchronization optimization policies. The idea is to generate a conditional acquire or release construct at all of the sites that may acquire or release a lock in any of the synchronization optimization policies. Each site has a flag that controls whether it actually executes the construct; each acquire or release site tests its flag

to determine if it should acquire or release the lock. In this scenario, the generated code switches policies by changing the values of the flags. The advantage of this approach is the guarantee of no code growth; the disadvantage is the residual flag checking overhead at each conditional acquire or release site.

4.3 Measuring the Overhead of Each Policy

To choose the policy with the least overhead, the generated code must first measure the overhead. The compiler instruments the code to collect three measurements:

—**Locking Overhead:** The generated code computes the locking overhead by counting the number of times that the computation acquires and releases a lock. This number is computed by incrementing a counter every time the computation acquires a lock. The locking overhead is simply the time required to acquire and release a lock times the number of times the computation acquires a lock.

—**Waiting Overhead:** The current implementation uses spin locks. The hardware exports a construct that allows the computation to attempt to acquire a lock; the return value indicates whether the lock was actually acquired. To acquire a lock, the computation repeatedly executes the hardware lock acquire construct until the attempted acquire succeeds. The computation increments a counter every time an attempt to acquire a lock fails. The waiting overhead is the time required to attempt, and fail, to acquire a lock times the number of failed acquires.

—**Execution Time:** The amount of time that the computation spends executing code from the application. This time is measured by reading the timer when a processor starts to execute application code, then reading the timer again when the processor finishes executing application code. The processor then subtracts the first time from the second time, and adds the difference to a running sum. As measured, the execution time includes the waiting time and the time spent acquiring and releasing locks. It is possible to subtract these two sources of overhead to obtain the amount of time spent performing useful computation.

Together, these measurements allow the compiler to evaluate the total overhead of each synchronization optimization policy. The total overhead is simply the lock overhead plus the waiting overhead divided by the execution time. The total overhead is therefore always between zero and one. The compiler uses the total overhead to choose the best synchronization optimization policy — the policy with the lowest overhead is the best.

One potential concern is the instrumentation overhead, which consists primarily of the operations that count the number of times each processor acquires and releases a lock. The operations that count the number of times that a processor attempted to acquire a lock and failed have little impact on the execution, because they occur during an interval when the processor would otherwise be waiting idle for another processor to release the lock.

We experimentally measured the impact of the instrumentation overhead on our set of benchmark applications by generating versions of the applications that use a single, statically chosen, synchronization optimization policy. We then executed these versions with and without the instrumentation. The performance differences between the instrumented and uninstrumented versions were very small, which indicates that the instrumentation overhead had little or no impact on the overall performance.

4.4 Choosing Sampling and Production Intervals

The sizes of the target sampling and production intervals can have a significant impact on the overall performance of the generated code. Excessively long sampling intervals may degrade the performance by executing non-optimal versions of the code for a long time. But if the sampling interval is too short, it may not yield an accurate measurement of the overhead. In the worst case, an inaccurate overhead measurement may cause the production phase to use the wrong synchronization optimization policy.

We expect the minimum absolute length of the sampling interval to be different for different applications. In practice, we have had little difficulty choosing default values that work well for our applications. In fact, it is possible to make the target sampling intervals very small for all of our applications — the minimum effective sampling intervals are large enough to provide overhead measurements that accurately reflect the relative overheads in the production phases.

To achieve good performance, the production phase must be long enough to profitably amortize the cost of the sampling phase. In practice, we have found that the major component of the sampling cost is the time spent executing the non-optimal versions.

In our current implementation of dynamic feedback, the length of the parallel section may also limit the performance. Our current implementation always executes a sampling phase at the beginning of each parallel section. If a parallel section does not contain enough computation for a production phase of the desired length, the computation may be unable to successfully amortize the sampling overhead. It should be possible to eliminate this potential problem by generating code that allows sampling and production intervals to span multiple executions of the parallel phase. This code would still maintain separate sampling and production intervals for each parallel section, but allow the intervals to contain multiple executions of the section.

In practice, we have had little difficulty choosing target production intervals that work well for our applications. All of our applications perform well with a fixed target production intervals that range from five to 1000 seconds.

4.5 Early Cut Off and Policy Ordering

In many cases, we expect that the individual sources of overhead will be either monotonically nondecreasing or monotonically nonincreasing across the set of possible implementations. The locking overhead, for example, never increases as the policy goes from Original to Bounded to Aggressive. The waiting overhead, on the other hand, should never decrease as the policy goes from Original to Bounded to Aggressive. These properties suggest the use of an early cut off to limit the number of sampled policies. If the Aggressive policy generates very little waiting overhead or the Original policy generates very little locking overhead, there is no need to sample any other policy.

It may therefore be possible to improve the sampling phase by trying extreme policies first, then going directly to the production phase if the overhead measurements indicate that no other policy would do significantly better. It may also be possible to improve the sampling phase by ordering the policies. The generated code could sample a given policy first if it has done well in the past. If the measured overhead continued to be acceptable, the generated code could go directly to the production phase.

4.6 Limitations

In this article, we present an application of dynamic feedback to a specific problem: the choice of the best synchronization optimization policy. In the context of our benchmark applications, this problem has a set of characteristics that make it particularly suitable for the use of dynamic feedback. First, we were able to identify a small number of potential policies that were suitable for the synchronization optimization problem. The final executable contains code compiled with each different policy. For problems with a large number of potential policies, it is clearly infeasible to include code for all of the policies in the shipped executable. As discussed in Section 4.2, it may be possible in some cases to generate parameterized code that would implement different policies depending on the setting of the parameters. In general, however, it would be necessary to generate the code for each policy dynamically, which would add to the overhead of using dynamic feedback.

The sampling phases in the dynamic feedback algorithm execute different computations with different optimal execution times. This raises the question of how to compare performance results from different computations with different optimal execution characteristics. Locks support a performance measure (the sum of the locking and waiting overheads) that can be used to compute the overhead of each policy as a percentage of the running time of the computation. Because these overhead measurements factor out the running times of the computations, they can be used to compare the performance impact of the different policies even when the policies were used for different computations. In general, however, it may not be clear how to derive a performance metric with this property.

In all of our applications, the locking frequency is small enough and the overhead of acquiring and releasing a lock is large enough so that it is possible to instrument the locking code without significantly perturbing the performance or execution characteristics of the sampling code. In general, however, instrumentation code may unacceptably perturb the computation. In some situations, zero-overhead hardware instrumentation such as performance counters may provide an acceptable alternative [Anderson et al. 1997].

In all of our applications, the concurrency is exploited within parallel loops that contain synchronized atomic operations. Parallel loops provide a convenient way to set up the boundaries of the sampling and production phases: the compiler can simply check for the expiration of the phase at the beginning of each loop iteration.[5] In our applications, the loop iterations are large enough to profitably amortize the checking overhead, but small enough to quickly detect the expiration of sampling and production phases. In general, the issue of where to place the code that checks for the expiration of phases may complicate the application of dynamic feedback to different optimization problems.

In all of our applications, the best policy is relatively constant for each parallel loop. So the approach of sampling to find the best policy, then using the best policy for a relatively long production phase, works well for our applications. For applications in which the best policy changes more quickly, it would be necessary to sample more frequently. Because increasing the sampling frequency also increases the sampling overhead, it is possible to imagine applications for which it is simply not possible to sample fast enough to profitably adapt to changes in the best policy. This raises the question of how to choose the best sampling and production intervals. In most situations, we expect the smallest sampling

[5] As described in Section 4.1, it would be straightforward to generalize this approach to check also at procedure boundaries. This might be important in applications that use recursion as a primary control flow construct instead of loops.

frequency that profitably amortizes the sampling overhead to be significantly smaller than the largest sampling frequency that adapts quickly enough to changes in the best policy.

In extreme cases, it might be possible to use a second-order application of dynamic feedback to choose an acceptable sampling frequency. In these scenarios, the compiler would generate code that accepts the sampling frequency as a parameter. A second-order sampling algorithm would test different values of the sampling frequency to find one with an acceptable trade-off between the adaptation latency and the sampling overhead.

Another issue related to the sizes of the sampling and production intervals is that the production interval must be long enough to profitably amortize the time wasted while sampling a policy with poor performance. Section 5 presents a theoretical analysis that can be used to guide the choice of sampling and production interval lengths, but this analysis is valid only under some fairly restrictive assumptions. In practice, some of our applications sample policies that deliver almost no useful work. Even for these applications, it was not difficult to obtain a production interval long enough to amortize the sampling overhead.

5. DYNAMIC SELECTION OF SAMPLING AND PRODUCTION INTERVALS

In this section we present a theoretical analysis of the performance of dynamic feedback. We start by observing that if no constraint is imposed on how fast the overhead of each policy may change, it is difficult to obtain any meaningful optimality result for any sampling algorithm — the overhead of each policy may change dramatically right after the sampling phase. To simplify the analysis, we assume that the best policy does not change during the production intervals. In this situation, the worst-case scenario for dynamic feedback is for the best policy to have no overhead at all, and for all of the other policies to perform no useful work. We analyze a slightly more general scenario in which there are N policies whose relative overheads do not change over time. While simple, this scenario provides insight into how to dynamically select the lengths of the sampling and production intervals. As part of this theoretical analysis, we derive a performance bound for dynamic feedback as compared to the optimal algorithm that always uses the best policy.

5.1 Definitions

We first establish some notation. There are N different policies, $p_0, \cdots, p_{N-1}$. Each variable s_i denotes the effective sampling interval for the policy p_i. The total length S of the sampling phase is therefore the sum of the sampling intervals for the sampled policies, i.e., $S = \sum_{i=0}^{N-1} s_i$. The length of the production interval is denoted by P.

The computation starts with a sampling phase. During this phase, the dynamic feedback algorithm executes each of the N policies for a sampling interval s_i to derive overhead measurements $o_0, \cdots, o_{N-1}$ for each of the N policies. The overhead is the proportion of the total execution time spent acquiring and releasing locks or waiting for other processors to release locks. The overhead therefore varies between zero (if the computation never acquires a lock) and one (if the computation performs no useful work), i.e. $0 < o_i < 1$.

We now define a precise way to measure the amount of useful work that a given policy delivers during a given period of time.

DEFINITION 1. *The amount of useful work delivered by a policy p_i over a period of time T is denoted by* $\texttt{Work}_{p_i}^T$ *where* $\texttt{Work}_{p_i}^T$ *is defined as*

$$\texttt{Work}_{p_i}^T = T(1 - o_i) \tag{1}$$

Conversely, we define the performance of a given policy as the amount of time taken to deliver W units of work. Given the definition above for the amount of useful work, finding the time it takes for a given policy to deliver W units of work amounts to finding the time T for which $T(1 - o_i)$ equals W. The next definition formalizes this notion.

DEFINITION 2. *The performance of a given policy p_i for a given amount of work W is defined to be the amount of time $\texttt{Time}_{p_i}^{W}$ that p_i takes to deliver W units of work. Given an amount of work W, the following equation provides the value of $\texttt{Time}_{p_i}^{W}$:*

$$\texttt{Time}_{p_i}^{W} = \frac{W}{(1 - o_i)} \tag{2}$$

We now define how to precisely compare the performance of two policies. Basically, we compare the performance of two policies by computing the ratio of the time they require to deliver a given amount of work. The definition below formalizes this notion.

DEFINITION 3. *Policy p_i is at most ϵ worse than policy p_j for a given amount of work W if*

$$\frac{\texttt{Time}_{p_i}^{W}}{\texttt{Time}_{p_j}^{W}} \leq (1 + \epsilon) \tag{3}$$

5.2 Performance Analysis

We now analyze the performance of dynamic feedback. We first derive the performance of dynamic feedback algorithm, then compare the performance of this algorithm with that of the optimal algorithm that always executes the policy with the lowest overhead. Based on this comparison, we determine the production interval P for which the dynamic feedback algorithm is guaranteed to be at most ϵ worse than the optimal algorithm.

Without loss of generality, we assume that policy p_0 is the policy with the lowest sampled overhead. The dynamic feedback algorithm therefore executes policy p_0 during the production interval P. We further assume that the values measured during the sampling phase accurately reflect the actual overheads at the start of the production phase, and that the measured overheads do not change during the sampling or production phases.

Under these assumptions, the worst-case scenario is for the dynamic feedback algorithm to complete the work at the very end of a sampling phase. In other words, the dynamic feedback algorithm completes $M + 1$ sampling phases but only M production phases. This is the worst-case scenario because it maximizes the amount of time that the dynamic feedback algorithm spends executing suboptimal policies. We derive the time it would take for the optimal algorithm to deliver the same amount of work, then compare the times taken by the two algorithms.

For this scenario, the amount of work performed by the dynamic feedback algorithm during the $M + 1$ sampling phases and M production phases is given by:

$$\texttt{Work}_{df} = (M + 1) \sum_{i=0}^{N-1} (1 - o_i) s_i + M(1 - o_0) P \tag{4}$$

The corresponding amount of time required to deliver this amount of work is simply the combined time of the $M + 1$ sampling phases and M production phases.

$$\texttt{Time}_{df} = (M+1)\sum_{i=0}^{N-1} s_i + MP \tag{5}$$

The time required for the optimal algorithm to deliver the same amount of work is:

$$\texttt{Time}_{opt} = \frac{\texttt{Work}_{df}}{(1-o_0)} = (M+1)\sum_{i=0}^{N-1} \frac{(1-o_i)}{(1-o_0)} s_i + MP \tag{6}$$

The relative performance of the two algorithms, as defined above, is:

$$\frac{\texttt{Time}_{df}}{\texttt{Time}_{opt}} = \frac{(M+1)\sum_{i=0}^{N-1} s_i + MP}{(M+1)\sum_{i=0}^{N-1} \frac{(1-o_i)}{(1-o_0)} s_i + MP} \tag{7}$$

Expanding the factor s_i in the numerator first as $\left(\frac{(1-o_0+o_i-o_i)}{(1-o_0)}\right) s_i$, then as $\left(\frac{(1-o_i)}{(1-o_0)} + \frac{(o_i-o_0)}{(1-o_0)}\right) s_i$, we obtain the following intermediate equation:

$$\frac{\texttt{Time}_{df}}{\texttt{Time}_{opt}} = \frac{(M+1)\sum_{i=0}^{N-1} \left(\frac{(1-o_i)}{(1-o_0)} + \frac{(o_i-o_0)}{(1-o_0)}\right) s_i + MP}{(M+1)\sum_{i=0}^{N-1} \frac{(1-o_i)}{(1-o_0)} s_i + MP} \tag{8}$$

Distributing the summation across the two fractions in the numerator yields:

$$\frac{\texttt{Time}_{df}}{\texttt{Time}_{opt}} = \frac{(M+1)\sum_{i=0}^{N-1} \left(\frac{(1-o_i)}{(1-o_0)}\right) s_i + (M+1)\sum_{i=0}^{N-1} \left(\frac{(o_i-o_0)}{(1-o_0)}\right) s_i + MP}{(M+1)\sum_{i=0}^{N-1} \frac{(1-o_i)}{(1-o_0)} s_i + MP} \tag{9}$$

We can rearrange the the terms in the numerator to obtain the following equation:

$$\frac{\texttt{Time}_{df}}{\texttt{Time}_{opt}} = 1 + \frac{(M+1)\sum_{i=0}^{N-1} \frac{(o_i-o_0)}{(1-o_0)} s_i}{(M+1)\sum_{i=0}^{N-1} \frac{(1-o_i)}{(1-o_0)} s_i + MP} \tag{10}$$

Given a performance bound ϵ, we use Equation 10 above to derive the minimum value for the production interval P required to satisfy the performance bound.

$$\frac{\texttt{Time}_{df}}{\texttt{Time}_{opt}} \leq (1+\epsilon) \Rightarrow 1 + \frac{(M+1)\sum_{i=0}^{N-1} \frac{(o_i-o_0)}{(1-o_0)} s_i}{(M+1)\sum_{i=0}^{N-1} \frac{(1-o_i)}{(1-o_0)} s_i + MP} \leq (1+\epsilon) \tag{11}$$

Algebraic simplification of this inequality yields:

$$(M+1)\sum_{i=0}^{N-1} \frac{(o_i-o_0)}{(1-o_0)} s_i \leq (M+1)\sum_{i=0}^{N-1} \frac{(1-o_i)}{(1-o_0)} s_i \epsilon + MP\epsilon \tag{12}$$

We can divide both sides of this inequality by $M\epsilon$ to obtain the following inequality:

$$\frac{(M+1)}{M}\frac{\sum_{i=0}^{N-1}\frac{(o_i-o_0)}{(1-o_0)}s_i}{\epsilon} \leq \frac{(M+1)}{M}\sum_{i=0}^{N-1}\frac{(1-o_i)}{(1-o_0)}s_i + P \tag{13}$$

We can now rearrange terms and divide to obtain the final inequality:

$$P \geq \frac{(M+1)}{M}\left(\frac{\sum_{i=0}^{N-1}\frac{(o_i-o_0)}{(1-o_0)}s_i}{\epsilon} - \sum_{i=0}^{N-1}\frac{(1-o_i)}{(1-o_0)}s_i\right) \tag{14}$$

This inequality defines an inverse relationship between P and ϵ: the smaller the performance bound ϵ, larger the production interval P required to meet the performance bound. For infinite work, i.e., as $M \to \infty$, the asymptotic value of the performance bound ϵ_∞ for a given production interval P is:

$$\epsilon_\infty = \lim_{M\to\infty}\frac{\texttt{Time}_{df}}{\texttt{Time}_{opt}} - 1 = \frac{\sum_{i=0}^{N-1}\frac{(o_i-o_0)}{(1-o_0)}s_i}{\sum_{i=0}^{N-1}\frac{(1-o_i)}{(1-o_0)}s_i + P} \tag{15}$$

In our implementation of dynamic feedback, the compiler generates code that experimentally extracts the values for the sampling intervals s_i and sampled overheads o_i. In Section 6, we use this data to calculate asymptotic performance bounds for dynamic feedback as applied to our benchmark programs.

6. EXPERIMENTAL RESULTS

This section presents experimental results that characterize how well dynamic feedback works for three benchmark applications. The applications are Barnes-Hut, a hierarchical N-body solver [Barnes and Hut 1986], Water, which simulates water molecules in the liquid state [Singh et al. 1992], and String, which builds a velocity model of the soil between two oil wells [Harris et al. 1990]. Each application is a serial C++ program that performs a computation of interest to the scientific computing community. Barnes-Hut consists of approximately 1500 lines of code, Water consists of approximately 1850 lines of code, and String consists of approximately 2050 lines of code. We used our prototype compiler to parallelize each application. This parallelization is completely automatic — the programs contain no pragmas or annotations, and the compiler performs all of the necessary analyses and transformations. To compare the performance impact of the different synchronization optimization policies, we used compiler flags to obtain four different versions of each application. One version uses the Original policy, another uses the Bounded policy, another uses the Aggressive policy, and the final version uses dynamic feedback.

We report results for the applications running on a 16 processor Stanford DASH machine [Lenoski 1992] running a modified version of the IRIX 5.2 operating system. The programs were compiled using the IRIX 5.3 CC compiler at the `-O2` optimization level. We report the best results out of five runs for each application and evaluation parameter settings.

6.1 Stanford DASH Machine Characteristics

The Stanford DASH machine [Lenoski 1992] is a cache-coherent shared-memory multiprocessor. It uses a distributed directory-based protocol to provide cache coherence. It is

organized as a group of processing clusters connected by a mesh interconnection network. Each of the clusters is a Silicon Graphics 4D/340 bus-based multiprocessor. The 4D/340 system has four processing nodes, each of which contains a 33MHz R3000 processor, a R3010 floating-point co-processor, a 64KByte instruction cache, 64KByte first-level write-through data cache, and a 256KByte second-level write-back data cache. Each node has a peak performance of 25 VAX MIPS and 10 double-precision MFLOPS. Cache coherence within a cluster is maintained at the level of 16-byte lines via a bus-based snoopy protocol. Each cluster also includes a directory processor that snoops on the bus and handles references to and from other clusters. The directory processor maintains directory information on the cacheable main memory within that cluster that indicates which clusters, if any, currently cache each line.

The interconnection network consists of a pair of wormhole routed meshes, one for request messages and one for replies. The total bandwidth in and out of each cluster is 120 megabytes per second.

6.2 Barnes-Hut

Table II presents the execution times for the different versions of Barnes-Hut. Figure 4 presents the corresponding speedup curves. All experimental results are for an input data set of $16,384$ bodies. The static versions (Original, Bounded and Aggressive) execute without the instrumentation required to compute the locking or waiting overhead. The Dynamic version, the version that uses dynamic feedback, must contain this instrumentation because it uses the locking and waiting overhead measurements to determine the best synchronization optimization policy.[6]

Version	Processors					
	1	2	4	8	12	16
Serial	147.8	—	—	—	—	—
Original	217.2	111.6	56.59	32.61	20.76	15.64
Bounded	191.7	97.25	49.22	26.98	19.62	15.12
Aggressive	149.9	76.30	37.81	21.88	15.57	12.87
Dynamic	158.3	80.37	41.00	24.27	17.22	13.85

Table II. Execution Times for Barnes-Hut (seconds)

For this application, the synchronization optimization policy has a significant impact on the overall performance, with the Aggressive version significantly outperforming both the Original and the Bounded versions. The performance of the Dynamic version is quite close to that of the Aggressive version. A programmer manually tuning this application would select the Aggressive as the best static synchronization optimization policy.

Table III presents the locking overhead for the different versions of Barnes-Hut. The execution times are correlated with the locking overhead. For all versions except Dynamic, the number of executed acquire and release constructs (and therefore the locking overhead) does not vary as the number of processors varies. For the Dynamic version, the number of executed acquire and release constructs increases slightly as the number of processors

[6] Strictly speaking, the Dynamic version needs to execute instrumented code only during the sampling phase. But because the instrumentation overhead does not significantly affect the performance, the production phase simply executes the same instrumented code as the best version in the previous sampling phase. This approach inhibits code growth by eliminating the need to generate instrumented and uninstrumented versions of the code.

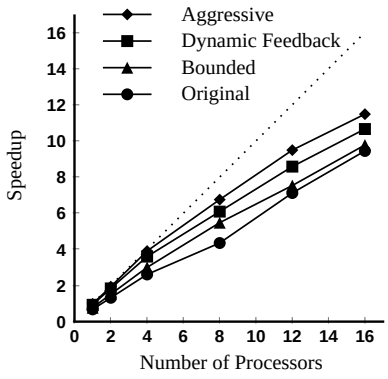

Fig. 4. Speedups for Barnes-Hut

increases. The numbers in the table for the Dynamic version are from an eight processor run.

Version	Executed Acquire And Release Pairs	Absolute Locking Overhead (seconds)
Original	15,471,682	77.4
Bounded	7,744,033	38.7
Aggressive	49,152	0.246
Dynamic	72,050	0.360

Table III. Locking Overhead for Barnes-Hut

Although the absolute performance varies with the synchronization optimization policy, the performance of the different versions scales at approximately the same rate. This indicates that the synchronization optimizations introduced no significant false exclusion. The reason that this application does not exhibit perfect speedup is that the compiler is unable to parallelize one section of the computation. At large numbers of processors the serial execution of this section becomes a bottleneck [Rinard and Diniz 1997].

To investigate how the overheads of the different policies change over time, we produced a version of the application with small target sampling and production intervals. We instrumented this version to print out the measured overhead at the end of each sampling interval. Figure 5 presents this data from an eight processor run in the form of a time-series graph for the main computationally intensive parallel section, the FORCES section. Our benchmark executes the FORCES section two times. The gap in the time series lines corresponds to the execution of a serial section of the code. Figure 5 shows that the measured overheads stay relatively stable over time.

We next discuss the characteristics of the application that relate to the minimum effective sampling interval for the FORCES section. The computation in this section consists of a single parallel loop. Table IV presents the mean section size, the number of iterations in the parallel loop, and the mean iteration size. The mean section size is the mean execution time of the FORCES section in the serial version, and is intended to measure the amount

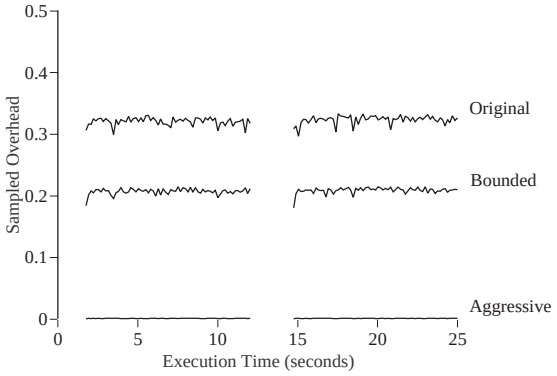

Fig. 5. Sampled Overhead for the Barnes-Hut FORCES Section on Eight Processors

of useful work in the section. We report the mean iteration size because the generated code checks for the expiration of the sampling and production intervals only at the granularity of the loop iterations. The loop iterations must therefore be large enough to amortize the expiration checks. For Barnes-Hut this is clearly the case — the cost of the expiration check is dominated by the 9 microsecond cost of reading the timer, which is negligible compared with the 4.2 millisecond mean iteration size. Tables X and XVI present the mean loop iteration sizes for Water and String, respectively. The cost of the expiration check is also negligible compared with the mean iteration sizes in these applications.

Mean Section Size	Number of Iterations	Mean Iteration Size
69.14 seconds	16,384	4.2 milliseconds

Table IV. Statistics for the Barnes-Hut FORCES Section

We used the version with small target sampling and production intervals to measure the minimum effective sampling intervals for each of the different synchronization optimization policies. In this version, the sampling and production intervals are as small as possible given the application characteristics — in other words, the actual intervals are the same length as the minimum effective sampling intervals. We instrumented this version to measure the length of each actual sampling interval, and used the data to compute the mean minimum effective sampling interval for each policy. Table V presents the data from an eight processor run. As expected, the mean minimum effective sampling intervals are larger than but still roughly comparable in size to the mean loop iteration size. The differences in the mean minimum effective sampling intervals are correlated with the differences in lock overhead. As the lock overhead increases, the amount of time required to execute each iteration also increases. Because none of the versions have significant waiting overhead, the increases in the amount of time required to execute each iteration translate directly into increases in the mean minimum effective sampling interval. Table V also presents the mean sampled overhead for each of the available policies on eight processors. We measured these overheads by setting the production interval to zero and saving all of the sampled overhead values.

Version	Mean Minimum Effective Sampling Interval (milliseconds)	Mean Sampled Overhead
Original	10.0	0.333
Bounded	7.8	0.216
Aggressive	6.5	0.002

Table V. Mean Minimum Effective Sampling Intervals for Barnes-Hut FORCES Section on Eight Processors

We next consider the impact of varying the target sampling and production intervals. For the performance numbers in Table II, the target sampling interval was set to 10 milliseconds and the target production interval was set to 1000 seconds. This target sampling interval was small enough to ensure that the minimum effective sampling interval, rather than the target sampling interval, determined the length of each actual sampling interval. A target production interval of 1000 seconds was long enough to ensure that each parallel section finished before it executed another sampling phase. The execution of each parallel section therefore consisted of one sampling phase and one production phase.

Table VI presents the execution times for the Barnes application running on eight processors for several combinations of target sampling and production intervals. Both performance figures are relatively insensitive to the variation in the target sampling and production intervals. Even when the target sampling and production intervals are identical (which means that the computation spends approximately three times as long in the sampling phase as in the production phase), the overall performance is never more than 11% slower than with the best combination of the sampling and production interval.

Target Sampling Interval (seconds)	FORCES Section Target Production Interval (seconds)				Complete Application Target Production Interval (seconds)			
	1	5	10	1000	1	5	10	1000
0.01	9.138	9.058	9.058	9.025	24.29	24.10	24.20	23.72
0.1	9.697	9.178	9.122	9.220	24.33	23.52	23.38	23.64
1.0	10.784	9.834	9.726	9.670	26.89	24.91	24.65	23.69

Table VI. Execution Times for Varying Production and Sampling Intervals for Barnes-Hut on Eight Processors (seconds)

In Section 5 we derived Equation 15, which provides the asymptotic performance bound ϵ_∞ of dynamic feedback given values for the overheads o_i, sampling intervals s_i, and production interval P. This equation is valid under the assumption that the overheads do not change during the production phase. As Figure 5 indicates, the overheads of the different policies are roughly constant for the FORCES section of Barnes-Hut. Table VII presents the calculated asymptotic performance bounds for the FORCES section of Barnes-Hut on eight processors as a function of the measured overheads and different production and sampling intervals.

For each policy p_i, we use the mean sampled overhead as o_i and the maximum of the target sampling interval and that policy's mean minimum effective sampling interval as the value for that policy's sampling interval s_i in the calculations. This choice reflects the fact that for some policies, it may not be possible to achieve a small target sampling interval. For the FORCES section of Barnes-Hut, all of the mean minimum effective sampling intervals are smaller than all of the target sampling intervals. This is not true for our other applications.

The asymptotic bounds numbers show that the asymptotic performance of the dynamic feedback algorithm is very close to that of the optimal algorithm. Even when the sampling and production intervals are the same, the asymptotic bound is less than 0.16.

Target Sampling Interval (seconds)	Application Section FORCES Target Production Interval (seconds) 1	5	10	1000
0.01	.005	.001	.001	.0000
0.1	.044	.010	.005	.0001
1.0	.158	.073	.044	.0001

Table VII. Asymptotic Performance Bound ϵ_∞ for Varying Production and Sampling Intervals for the FORCES Section of Barnes-Hut on Eight Processors

6.3 Water

Table VIII presents the execution times for the different versions of Water. Figure 6 presents the corresponding speedup curves. All experimental results are for an input data set of 512 molecules. The static versions (Original, Bounded and Aggressive) execute without the instrumentation required to compute the locking or waiting overhead. The Dynamic version needs the instrumentation to apply the dynamic feedback algorithm, so this version contains the instrumentation.

Version	Processors 1	2	4	8	12	16
Serial	165.8	—	—	—	—	—
Original	184.4	94.60	47.51	28.39	22.06	19.87
Bounded	175.8	88.36	44.28	26.42	21.06	19.50
Aggressive	165.3	115.2	88.45	79.18	75.16	73.54
Dynamic	165.4	88.76	44.29	27.20	21.60	20.54

Table VIII. Execution Times for Water (seconds)

For this application, the synchronization optimization policy has a significant impact on the overall performance. For one processor, the Aggressive version performs the best. As the number of processors increases, however, the Aggressive version fails to scale, and the Bounded version outperforms both the Aggressive and the Original versions. For this application, the Bounded policy is the best static synchronization optimization policy. As the performance results presented below indicate, false exclusion causes the poor performance of the Aggressive version. The performance of the Dynamic version is very close to the performance of the Bounded version, which exhibits the best performance.

Table IX presents the locking overhead for the different versions of Water. For the Original, Bounded, and Dynamic versions, the execution times are correlated with the locking overhead. For all versions except Dynamic, the number of executed acquire and release constructs (and therefore the locking overhead) does not vary as the number of processors varies. For the Dynamic version at two processors and above, the number of executed acquire and release constructs is very close to the Bounded version, with a slight increase as

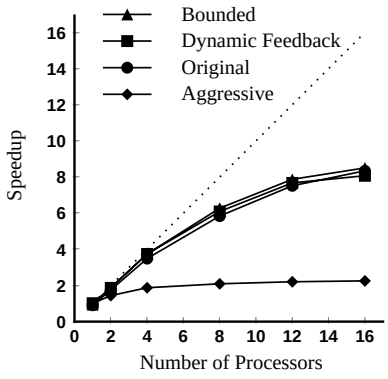

Fig. 6. Speedups for Water

the number of processors increases. At one processor, the Dynamic version executes approximately the same number of acquire and release constructs as the Aggressive version. The numbers in the table for the Dynamic version are from an eight processor run.

Version	Executed Acquire And Release Pairs	Absolute Locking Overhead (seconds)
Original	4,200,448	21.0
Bounded	2,099,200	10.5
Aggressive	1,577,980	7.9
Dynamic	2,119,840	10.6

Table IX. Locking Overhead for Water

We instrumented the parallel code to determine why Water does not exhibit perfect speedup. Figure 7 presents the *waiting proportion*, which is the proportion of time spent in waiting overhead.[7] These data were collected using program-counter sampling to profile the execution [Graham et al. 1982; Knuth 1971]. This figure clearly shows that waiting overhead is the primary cause of performance loss for this application, and that the Aggressive synchronization optimization policy generates enough false exclusion to severely degrade the performance.

Water has two computationally intensive parallel sections: the INTERF section and the POTENG section. Figures 8 and 9 present time-series graphs of the measured overheads of the different synchronization optimization policies. For the INTERF section, the generated code would be the same for the Bounded and Aggressive policies. The compiler therefore does not generate an Aggressive version, and the sampling phases execute only the Original and Bounded versions. A similar situation occurs in the POTENG section, except that in this case, the code would be the same for the Original and Bounded versions. As for

[7] More precisely, the waiting proportion is the sum over all processors of the amount of time that each processor spends waiting to acquire a lock held by another processor divided by the execution time of the program times the number of processors executing the computation.

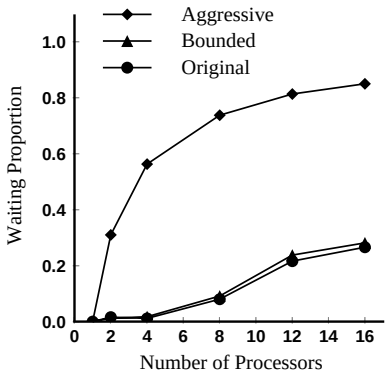

Fig. 7. Waiting Proportion for Water

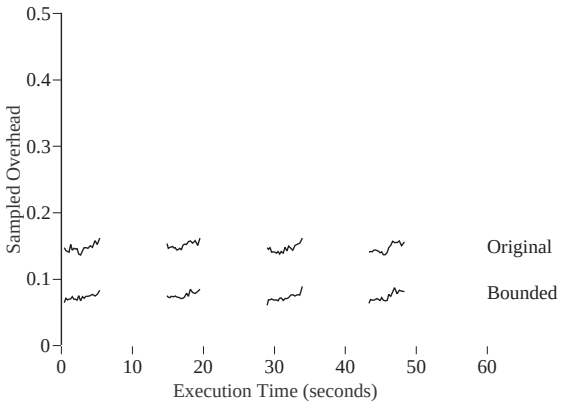

Fig. 8. Sampled Overhead for the Water INTERF Section on Eight Processors

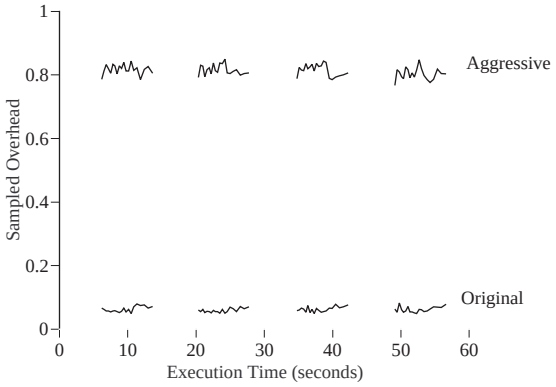

Fig. 9. Sampled Overhead for the Water POTENG Section on Eight Processors

Barnes-Hut, the overheads are relatively stable over time. The gaps in the time-series graphs correspond to the executions of other serial and parallel sections.

Table X presents the parallel section statistics for the INTERF and POTENG sections. Table XI presents the mean minimum effective sampling intervals and the mean sampled overheads for the two sections. As expected, the mean minimum effective sampling intervals for all of the versions except the Aggressive version in the POTENG section are larger than but still roughly comparable to the corresponding mean iteration sizes. The mean minimum effective sampling interval for the Aggressive version in the POTENG section is significantly larger than for the Original version. We attribute this difference to the fact that the Aggressive policy serializes much of the computation, which, as described in Section 4.1, increases the effective sampling interval.

Application Section					
INTERF			POTENG		
Mean Section Size	Number of Iterations	Mean Iteration Size	Mean Section Size	Number of Iterations	Mean Iteration Size
20.80 seconds	511	40.7 milliseconds	16.34 seconds	511	32.0 milliseconds

Table X. Statistics for Water

	Application Section			
	INTERF		POTENG	
Version	Mean Minimum Effective Sampling Interval (milliseconds)	Mean Sampled Overhead	Mean Minimum Effective Sampling Interval (milliseconds)	Mean Sampled Overhead
Original	93	0.152	59	0.226
Bounded	82	0.081	–	–
Aggressive	–	–	286	0.913

Table XI. Mean Minimum Effective Sampling Intervals and Mean Sampled Overheads for Water Sections on Eight Processors

For the performance numbers in Table VIII, the target sampling interval was set to 10 milliseconds and the target production interval was set to 1000 seconds. This combination ensured that the execution of each parallel section consisted of one sampling phase and one production phase.

Table XII presents the execution times for the INTERF and POTENG sections as well as the entire application running on eight processors for several combinations of target sampling and production intervals. For the INTERF section, all of the combinations yield approximately the same performance. We attribute this uniformity to the fact that the performance of the two versions in the section (the Original and Bounded versions) is not dramatically different.

Target Sampling Interval (seconds)	Application Section: INTERF, Target Production Interval (seconds)				Application Section: POTENG, Target Production Interval (seconds)				Application Section: Complete Application, Target Production Interval (seconds)			
	1	5	10	1000	1	5	10	1000	1	5	10	1000
0.01	3.67	3.54	3.51	3.55	3.32	2.62	2.64	2.65	29.54	26.25	26.11	26.34
0.1	3.73	3.53	3.53	3.57	3.07	2.69	2.71	2.72	28.78	26.49	26.55	26.70
1.0	3.71	3.67	3.68	3.67	4.18	3.48	3.48	3.49	33.12	30.26	30.24	30.12

Table XII. Mean Execution Times for Varying Production and Sampling Intervals for Water on Eight Processors (seconds)

As shown in Table XII, the performance of the POTENG section is quite sensitive to the choice of target sampling interval when the target production interval is either 1 or 5 seconds. Recall that there is a dramatic difference in this section between the performance of the Aggressive and Original versions. In this case, one would intuitively expect the performance to increase with increases in the target production interval and decrease with increases in the target sampling interval. We address the ways in which the data fail to conform to this expectation.

First, the execution times are virtually identical at target production intervals of 5, 10 and 1000 seconds. We attribute this uniformity to the fact that the execution of the POTENG section always terminates in less than 5 seconds. Extending the target production interval beyond 5 seconds therefore has no effect on the execution time.

Second, the execution times are virtually identical for a target production interval of 5 seconds and target sampling intervals of 0.01 and 0.1 seconds. We attribute these data to the fact that the execution of the POTENG section always terminates in less than 5 seconds and the fact that the minimum effective sampling interval for the Aggressive policy is greater than 0.1 seconds. Both of the executions in question consist of an Aggressive sampling interval whose length is the same in both executions, an Original sampling interval, then an Original production interval during which the section completes its execution. Both executions spend almost identical amounts of time executing the Aggressive and Original versions.

Finally, the execution time decreases for a target production interval of 1 seconds when the target sampling interval increases from 0.01 seconds to 0.1 seconds. The effect is caused by the fact that the minimum effective sampling interval of the Original version is smaller than 0.1 seconds, while the minimum effective sampling interval of the Aggressive version is larger than 0.1 seconds. The program therefore spends a larger proportion of the sampling phase executing the more efficient Original version with a target sampling

interval of 0.1 seconds than it does with a target sampling interval of 0.01 seconds. An effect associated with the end of the section exacerbates the performance impact. With a target sampling interval of 0.1 seconds, the section completes after two sampling phases and two production phases. With a target sampling interval of 0.01 seconds, the section performs less computation in the Original sampling intervals, and it does not complete until after it has executed a third Aggressive sampling interval. The net effect of the decrease in the target sampling interval is a significant increase in the amount of time spent executing the inefficient Aggressive sampling intervals.

Table XIII presents the asymptotic performance bounds for the INTERF and POTENG sections on eight processors. For the INTERF section, the asymptotic performance of the dynamic feedback algorithm is very close to the performance of the optimal algorithm. For the POTENG section, however, the asymptotic performance difference is significant when the target production interval is small. Two factors contribute to this behavior. First, the mean minimum effective sampling interval for the Aggressive policy in the POTENG section is approximately 0.29 seconds, which is quite large compared to the target production interval. Second, the mean sampled overhead of the Aggressive policy is 0.91 for the POTENG section. This policy delivers almost no useful work during its sampling phases. The result is that for small target production intervals, the application spends a significant amount of its time executing a policy that delivers almost no useful work.

	Application Section							
	INTERF				POTENG			
Target Sampling Interval (seconds)	Target Production Interval (seconds)				Target Production Interval (seconds)			
	1	5	10	1000	1	5	10	1000
0.01	.006	.001	.0007	0.00	.23	.05	.025	.0003
0.1	.007	.002	.0008	0.00	.22	.05	.025	.0003
1.0	.026	.011	.0065	0.01	.42	.15	.080	.0009

Table XIII. Asymptotic Performance Bound ϵ_∞ for Varying Production and Sampling Intervals for the INTERF and POTENG Sections of Water on Eight Processors

6.4 String

For String, the Bounded and Original policies produce the same parallel code. We therefore report performance results for only the Original, Aggressive, and Dynamic policies. Table XIV presents the execution times for the different versions of String. Figure 10 presents the corresponding speedup curves. All experimental results are for the Big Well input data set. The static versions (Original and Aggressive) execute without the instrumentation required to compute the locking or waiting overhead; the Dynamic version includes the instrumentation.

For String, the best synchronization optimization policy is the Original policy, as the Aggressive policy completely serializes the computation. The Aggressive version therefore fails to scale at all. The execution time of the Dynamic version is comparable to the execution time of the Original version, with a small loss of performance at 12 and 16 processors.

Table XV presents the locking overhead for the different versions of String. For the Dynamic version at two processors and above, the number of executed acquire and release

Version	Processors					
	1	2	4	8	12	16
Serial	2181.3	—	—	—	—	—
Original	2599.0	1289.4	646.7	331.9	223.9	172.3
Aggressive	2337.7	2313.5	2231.9	2244.3	2254.8	2260.9
Dynamic	2363.8	1295.5	653.5	342.5	241.3	194.9

Table XIV. Execution Times for String (seconds)

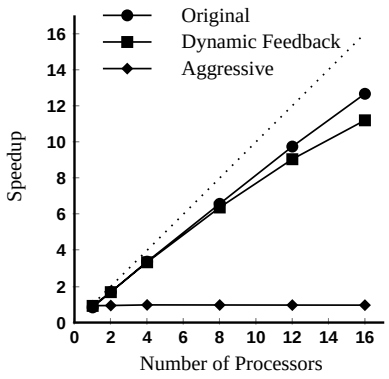

Fig. 10. Speedups for String

constructs is slightly less than in the Original version. The number also increases slightly as the number of processors increases. At one processor, the Dynamic version executes approximately six times fewer acquire and release constructs than the Original version. The numbers in the table for the Dynamic version are from an eight processor run.

Version	Executed Acquire And Release Pairs	Absolute Locking Overhead (seconds)
Original	30,286,596	151.43
Aggressive	2,313	0.01156
Dynamic	30,016,913	150.08

Table XV. Locking Overhead for String

We instrumented the parallel code to determine why String does not exhibit perfect speedup. Figure 11 presents the waiting proportion. This figure clearly shows that waiting overhead is the primary cause of performance loss for this application, and that the Aggressive synchronization optimization policy generates enough false exclusion to serialize the computation.

Figure 12 presents time-series graphs of the measured overheads of the different synchronization optimization policies for the main computationally intensive parallel section, the PROJFWD section. We collected these data by setting the target sampling and production intervals to one second, then instrumenting the code to print out the measured overhead

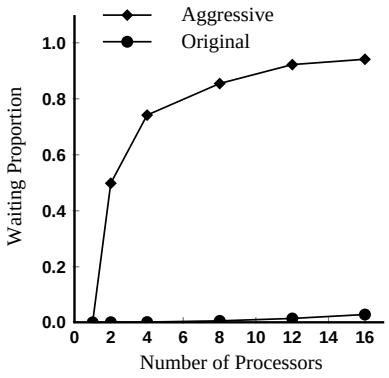

Fig. 11. Waiting Proportion for String

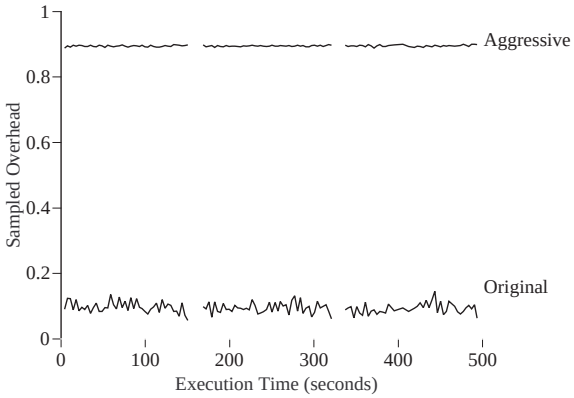

Fig. 12. Sampled Overhead for the String PROJFWD Section on Eight Processors

Mean Section Size	Number of Iterations	Mean Iteration Size
801 seconds	28,288	28.3 milliseconds

Table XVI. Statistics for the String PROJFWD Section

Version	Application Section PROJFWD	
	Mean Minimum Effective Sampling Interval (milliseconds)	Mean Sampled Overhead
Original	54	0.112
Aggressive	260	0.887

Table XVII. Mean Minimum Effective Sampling Intervals and Mean Sampled Overheads for String Sections

at the end of each sampling interval. As for Barnes-Hut and Water, the overheads are relatively stable over time. The gaps in the time-series graphs correspond to the executions of other serial and parallel sections.

Table XVI presents the parallel section statistics for the PROJFWD section. Table XVII presents the mean minimum effective sampling intervals and the mean sampled overheads for the available policies. The mean minimum effective sampling interval for the Original version is larger than but roughly comparable to the iteration size. As in the POTENG section of Water, the mean minimum effective sampling interval for the Aggressive version is significantly larger than for the Original version. The reason is the same: the Aggressive version serializes much of the computation.

For the performance numbers in Table XIV, the target sampling interval was set to 10 milliseconds and the target production interval was set to 1000 seconds. This combination ensured that the execution of each parallel section consisted of one sampling phase and one production phase. Table XVIII presents the execution times for the PROJFWD section running on eight processors and for the entire application for several combinations of target sampling and production intervals. As expected for an application with a section with dramatic efficiency differences between the versions, the performance increases with increases in the target production interval and decreases with increases in the target sampling interval.

	Application Code Section							
	PROJFWD				Complete Application			
Target Sampling Interval (seconds)	Target Production Interval (seconds)				Target Production Interval (seconds)			
	1	5	10	1000	1	5	10	1000
0.01	140.6	117.1	114.7	112.54	434.0	365.4	357.8	352.4
0.1	144.7	118.3	114.3	112.69	445.3	368.2	356.1	348.4
1.0	165.5	131.1	121.7	112.96	507.7	405.9	377.8	352.1

Table XVIII. Execution Times for Varying Production and Sampling Intervals for String on Eight Processors (seconds)

Table XIX presents the asymptotic performance bounds for the PROJFWD section on eight processors. For reasons similar to the POTENG section in Water, the asymptotic bound is fairly large when the target production interval is small. Two factors contribute to this behavior. First, the mean minimum effective sampling interval for the Aggressive policy is approximately 0.26 seconds, which is quite large compared to the target production interval. Second, the mean sampled overhead of the Aggressive policy is 0.88. This policy delivers almost no useful work during its sampling phases. The result is that for small target production intervals, the application spends a significant amount of its time executing a policy that delivers almost no useful work.

6.5 Discussion

For each application, the best static synchronization optimization policy is different from that of the other two applications. Furthermore, the performance differences are significant — at 16 processors, the best version of Barnes-Hut is approximately 20% faster than the worst; for Water, the best is more than three times faster than the worst; for String, the best is more than ten times faster than the worst. In all of these cases, dynamic feedback

Target Sampling Interval (seconds)	Application Code Section PROJFWD Target Production Interval (seconds) 1	5	10	1000
0.01	.21	.05	.023	.0002
0.1	.20	.04	.022	.0002
1.0	.41	.14	.078	.0008

Table XIX. Asymptotic Performance Bound ϵ for Varying Production and Sampling Intervals for PROJFWD Section of String on Eight Processors

allows the Dynamic version to exhibit performance that is not only very close to that of the best static policy, but also almost always better than that of the next best static policy. The compiler can therefore automatically generate robust code that performs well in a variety of environments, which eliminates the need for the programmer to manually tune the program to use the best synchronization optimization policy.

7. RELATED WORK

Many researchers have developed systems that collect information about the dynamic characteristics of programs, then use that information to improve the performance. We discuss several approaches: profiling, adaptive execution techniques, dynamic type feedback techniques for improving the performance of object-oriented languages, and dynamic techniques for parallelizing loops. We also discuss dynamic compilation and related work in synchronization optimizations.

7.1 Profiling

Profiling is a standard way to obtain information about the dynamic characteristics of a program. In this approach, the program is instrumented, then executed to collect profiling data. The program can then be recompiled, with the profiling data used to guide policy decisions in the compiler.

A fundamental issue in profiling is the level of granularity and the specific mechanism used to collect data. Some profilers modify the binary source code to insert instructions that increment counter variables [Smith 1991]. Other profilers periodically interrupt the execution of the binary to sample the program counter [Anderson et al. 1997; Zhang et al. 1997; Graham et al. 1982].

Profiling has been used to guide decisions to inline procedures in C programs [Chang et al. 1992], to drive instruction scheduling algorithms [Chen et al. 1994], to help place code so as to minimize the impact on the memory hierarchy [Pettis and Hansen 1990], to minimize the overhead associated with cache coherency in CC-NUMA machines [Chilimbi and Larus 1994], to aid in register allocation [Morris 1991; Wall 1986], and to direct the compiler to frequently executed parts of the program so that the compiler can apply further optimizations [Fernandez 1995; Anderson et al. 1997].

Brewer [Brewer 1995] describes a system that uses statistical modeling to automatically predict which algorithm will work best for a given combination of input and hardware platform. The different algorithms are implemented by hand, not automatically generated from a single specification. The system uses profiling to characterize the performance of the different algorithms on the different hardware platforms.

A primary difference between dynamic feedback and standard profile-based approaches is that dynamic feedback can adapt dynamically to the current execution environment, rather than generating code based on the assumption that the environment will always be similar to the environment in the profiling run of the program. Dynamic feedback can also adjust to changes that occur within a single execution of the program. Profile-based approaches typically collect a single aggregate set of measurements for the entire execution, and can therefore miss environment changes that take place within a single execution.

The Morph system [Zhang et al. 1997], however, is an exception: it uses profiling to dynamically optimize the generated code. Morph collects trace samples to find frequently executed paths. The instrumented binary code uses this profiling data to dynamically apply several low-level machine specific transformations such as branch alignment and procedure code layout.

Another important difference between profile-based approaches and dynamic feedback is that profile-based approaches usually use the profile information to statically choose one or two policies that should work well. The generated code is compiled only with those policies, and there is little if any code growth. Our implementation of dynamic feedback, on the other hand, produces code compiled with all of the different policies. For some optimizations, there are an enormous number of possible policies. For example, trace scheduling [Fisher 1981] optimizes frequently-executed paths in the program. In effect, each path through the program corresponds to a different policy; it is clearly infeasible to produce an executable that contains all of the different paths.

7.2 Adaptive Execution Techniques

Many other researchers have recognized the need to use dynamic performance data to optimize the execution of programs [Dubnicki and LeBlanc 1992; Cox and Fowler 1993; Romer et al. 1994; Saavedra and Park 1996; Kim and Vaidya 1996; Falsafi and Wood 1997].

Some of the adaptive execution approaches described in the literature can be viewed as using control-theoretic approaches to select the values of variables that control the behavior of a parameterized algorithm. Typically, the programmer defines a set of observable variables and a feedback function that uses the observed values to generate new values for the control variables. Changes in the values of the observable variables propagate through the feedback function to change the control variables, and the program responds by modifying its behavior. Ideally, the observable variables, control variables, and feedback function are defined so that the program maximizes its performance across a range of dynamic environments. Saavedra and Park describe the application of such an adaptive execution technique to determine, at run time, the best value for the prefetch distance in an algorithm that prefetches data accessed by a loop [Saavedra and Park 1996].

Fox et. al. describe a manual experiment in which communication servers use current values of available bandwidth, client processing capabilities, and display resolution to dynamically change the representation of the data to send to the client [Fox et al. 1996]. Although full automation is suggested, the authors do not describe a clear feedback mechanism and adaptation strategy.

In the context of consistency protocols for distributed shared-memory machines, researchers have developed systems that measure the behavior of the program to gather statistics about the pages that the program accesses [Kim and Vaidya 1996; Falsafi and

Wood 1997]. These systems use the gathered statistics to dynamically select a memory consistency protocol for each page.

The SCI standard uses a simple adaptive cache consistency scheme [IEEE 1993]. The implementation maintains a list of the nodes that hold references to shared pages. If exactly two nodes reference a shared page, SCI uses an update protocol — each modification by either of the nodes is reflected in the other node's cache. But if more than two nodes reference a page, SCI reverts to an invalidation scheme. At a different granularity, Dubnicki and LeBlanc [Dubnicki and LeBlanc 1992] describe a scheme to match the granularity of sharing of cache lines and either split or merge data blocks across caches based on recent access patterns.

All of these systems use techniques similar to dynamic feedback in that the memory consistency protocol for a given page or the cache line granularity can change dynamically if the application changes the way it accesses the corresponding memory. Another similarity is the assumption that the behavior of the system in the recent past can be used to predict its behavior in the near future.

7.3 Dynamic Dispatch Optimizations

In object-oriented languages, the method that is invoked at a given call site depends on the dynamic class of the receiver object. The same call site may therefore invoke many different methods; the algorithm that determines which method to invoke is called the dynamic dispatch algorithm. Researchers have proposed several adaptive optimizations for improving the efficiency of dynamic dispatch. The standard mechanism is to collect data that indicates which methods tend to be invoked from which call sites, then to insert a type test that checks for common types first [Chambers and Ungar 1989]. Profiling has been used in this context to predict the most frequently occurring class of the receiver object at a given call site [Grove et al. 1995]. This information is then used to drive optimizations that inline methods based on predictions about the class of the receiver.

Dynamic type feedback is designed to direct the compiler's attention to parts of the program that would benefit from optimization [Hölzle and Ungar 1994]. Once a method has been optimized, the generated code continues to collect data that can be used to drive further optimizations and reverse poor implementation choices. In this sense, dynamic feedback is similar to dynamic type feedback in that both techniques generate code that dynamically adapts to its execution environment.

7.4 Run-Time Analysis and Speculative Execution

In certain circumstances, a lack of statically available information may prevent the compiler from parallelizing the program. Several systems address this problem by parallelizing programs dynamically using information that is available only as the program runs. The inspector/executor approach dynamically analyzes the values in index arrays to automatically parallelize computations that access irregular meshes [Leung and Zahorjan 1993; Saltz et al. 1991]. The Jade implementation dynamically analyzes how tasks access data to exploit the concurrency in coarse-grain parallel programs [Rinard et al. 1992]. Speculative approaches optimistically execute loops in parallel, rolling back the computation if the parallel execution violates the data dependences [Rauchwerger and Padua 1995]. Predicated analysis can be used to dynamically determine if it is possible to parallelize a given loop [Moon et al. 1998].

A major difference between dynamic feedback and these run-time techniques is that

dynamic feedback is designed to automatically choose between several implementations that deliver the same functionality. Each implementation is equally valid, and may very well perform the best in the current environment. In all of the run-time techniques, the goal is clearly to parallelize the computation, but the compiler simply lacks the information necessary to do so. It must therefore postpone the decision to apply the optimization until run-time, when the information is available.

7.5 Dynamic Compilation

Dynamic compilation systems enable the generation of code at run time [Auslander et al. 1996; Engler 1996; Lee and Leone 1996]. Because delaying the compilation until run time provides the compiler with information about the concrete values of input parameters, the compiler may be able to generate more efficient code. Existing research has focused on providing efficient mechanisms for dynamic compilation.

We see dynamic compilation as one way to generate the different implementations that dynamic feedback samples to find a best implementation. The advantage would be the elimination of potential code growth — the memory used to hold the generated code can be deallocated if the code will not be executed for a significant period of time. The compiler could dynamically regenerate the code when the dynamic feedback algorithm needs to sample its performance.

The major drawback would be the overhead required to perform the compilation dynamically. This overhead would become less of a concern if the program executed sampling phases very infrequently — the dynamic compilation overhead would be amortized away by the long production phases.

7.6 Synchronization Optimizations

This article presents research that applies dynamic feedback to the problem of choosing the best synchronization policy. Our previous research produced analyses and transformations for reducing the synchronization overhead [Diniz and Rinard 1998; 1997]. As part of this research, we also developed the different synchronization optimization policies discussed in this article. Plevyak, Zhang and Chien have developed a similar synchronization optimization technique, *access region expansion*, for concurrent object-oriented programs [Plevyak et al. 1995]. Because access region expansion is designed to reduce the overhead in sequential executions of such programs, it does not address the trade off between lock overhead and waiting overhead. The goal is simply to minimize the lock overhead.

Lim and Agarwal [Lim and Agarwal 1994] developed a *reactive* synchronization mechanism for synchronization operations in multiprocessors. The basic idea is to change the implementation of the locking constructs based on the observed contention. This reactive synchronization mechanism resembles dynamic feedback in that it uses a dynamic selection mechanism to optimize the synchronization. A key difference is that a program that uses reactive synchronization does not change its synchronization granularity. Reactive synchronization is designed to choose the most efficient implementation for each individual locking construct given the current amount of contention for the lock. The optimizations described in this paper, on the other hand, improve the synchronization performance by eliminating locking constructs. Dynamic feedback mediates the resulting trade-off between false exclusion and locking overhead.

8. CONCLUSION

This article presents a technique, dynamic feedback, that enables computations to adapt dynamically to different execution environments. A compiler that uses dynamic feedback produces several different versions of the same source code; each version uses a different optimization policy. Dynamic feedback automatically chooses the most efficient version by periodically sampling the performance of the different versions.

We have implemented dynamic feedback in the context of a parallelizing compiler for object-based programs. The generated code uses dynamic feedback to automatically choose the best synchronization policy. For this particular optimization problem, our experimental results show that dynamic feedback enables the compiler to automatically generate code that exhibits performance comparable to that of code that has been manually tuned to use the best synchronization optimization policy. Issues that must be addressed when using dynamic feedback for other optimizations include choosing an appropriate set of optimization policies, finding points in the program where it is safe to switch policies, determining an appropriate frequency for checking for the expiration of sampling and production intervals, determining effective sampling and production target intervals, and deriving a performance metric that allows the system to compare the performance of different policies when the policies are used for different computations.

As systems become more complex, developers will find it increasingly difficult to statically choose single policies that give good performance in all situations. The emerging paradigm of mobile programs highlights the potential difficulties: mobile programs will be expected to run well on a wide range of systems, many of which will be simply unavailable during the development and compilation process. One way to attack this problem is to use techniques that deliver good performance by dynamically measuring and adapting to the characteristics of the current environment. Dynamic feedback is an example of such a technique.

ACKNOWLEDGMENTS

We wish to thank the anonymous referees for their many useful comments and suggestions.

REFERENCES

AMARASINGHE, S. P. AND LAM, M. S. 1993. Communication optimization and code generation for distributed-memory machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Program Language Design and Implementation.* ACM Press, New York, NY, 126–138.

ANDERSON, J., BERC, L., DEAN, J., GHEMAWAT, S., HENZINGER, M., LEUNG, S., SITES, R., VANDEVOORDE, M., WALDSPURGER, C., AND WEIHL, W. E. 1997. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems 15,* 4 (Nov.), 357–390.

ANDERSON, J. M. AND LAM, M. S. 1993. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Program Language Design and Implementation.* ACM Press, New York, NY, 112–125.

AUSLANDER, J., PHILIPOSE, M., CHAMBERS, C., EGGERS, S., , AND BERSHAD, B. 1996. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Program Language Design and Implementation*. ACM Press, New York, NY, 149–159.

BARNES, J. AND HUT, P. 1986. A hierarchical O(NlogN) force calculation algorithm. *Nature 324,* 4 (Dec.), 446–449.

BREWER, E. 1995. High-level optimization via automated statistical modeling. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM Press, New York, NY, 80–91.

CHAMBERS, C. AND UNGAR, D. 1989. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN '89 Conference on Program Language Design and Implementation*. ACM Press, New York, NY, 146–160.

CHANG, P., MAHLKE, S., CHEN, W., AND HWU, W. 1992. Profile-guided automatic inline expansion for C programs. *Software—Practice and Experience 22*, 5 (May), 349–369.

CHEN, W., MAHLKE, S., WARTER, N., ANIK, S., AND HWU, W. 1994. Profile-assisted instruction scheduling. *International Journal of Parallel Programming 22*, 2 (Apr.), 151–181.

CHILIMBI, T. AND LARUS, J. 1994. Cachier: A tool for automatically inserting cico annotations. In *Proceedings of the 1994 International Conference on Parallel Processing*. CRC Press Inc., Boca Raton, FL, II 89–98.

COX, A. AND FOWLER, R. 1993. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the 20th International Symposium on Computer Architecture*. ACM Press, New York, NY, 98–108.

DINIZ, P. AND RINARD, M. 1997. Synchronization transformations for parallel computing. In *Proceedings of the Twenty-fourth Annual ACM Symposium on the Principles of Programming Languages*. ACM Press, New York, NY, 187–200.

DINIZ, P. AND RINARD, M. 1998. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *Journal of Parallel and Distributed Computing 49*, 2 (Mar.), 218–244.

DUBNICKI, C. AND LEBLANC, T. 1992. Adjustable block size coherent caches. In *Proceedings of the 19th International Symposium on Computer Architecture*. ACM Press, New York, NY, 170–180.

ENGLER, D. 1996. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN '96 Conference on Program Language Design and Implementation*. ACM Press, New York, NY, 160–170.

FALSAFI, B., LEBECK, A., REINHARDT, S., SCHOINAS, I., HILL, M., LARUS, J., ROGERS, A., AND WOOD, D. 1994. Application-specific protocols for user-level shared memory. In *Proceedings of Supercomputing '94*. IEEE Computer Society Press, Los Alamitos, CA, 380–389.

FALSAFI, B. AND WOOD, D. 1997. Reactive-NUMA: A design for unifying S-COMA and CC-NUMA. In *Proceedings of the 24th International Symposium on Computer Architecture*. ACM Press, New York, NY, 229–240.

FERNANDEZ, M. 1995. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Program Language Design and Implementation*. ACM Press, New York, NY, 103–115.

FISHER, J. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers C-30*, 7 (July), 478–490.

FOX, A., GRIBBLE, S., BREWER, E., AND AMIR, E. 1996. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, New York, NY, 160–173.

FREUDENBERGER, S., SCHWARTZ, J., AND SHARIR, M. 1983. Experience with the SETL optimizer. *ACM Transactions on Programming Languages and Systems 5*, 1 (Jan.), 26–45.

GRAHAM, S., KESSLER, P., AND MCKUSICK, M. 1982. gprof: a call graph execution profiler. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*. ACM Press, New York, NY, 120–126.

GROVE, D., DEAN, J., GARRET, C., AND CHAMBERS, C. 1995. Profile-guided receiver class prediction. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM Press, New York, NY, 108–123.

GUPTA, M. AND BANERJEE, P. 1992. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems 3*, 2 (Mar.), 179–193.

HARRIS, J., LAZARATOS, S., AND MICHELENA, R. 1990. Tomographic string inversion. In *60th Annual International Meeting, Society of Exploration and Geophysics, Extended Abstracts*. 82–85.

HÖLZLE, U. AND UNGAR, D. 1994. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN '94 Conference on Program Language Design and Implementation*. ACM Press, New York, NY, 326–336.

IEEE 1993. *IEEE Standard for Scalable Coherent Interface (SCI)*. IEEE, 345 East 47th Street, New York.

KEMMERER, R. AND ECKMANN, S. 1985. UNISEX: a UNIx-based Symbolic EXecutor for Pascal. *Software—Practice and Experience 15*, 5 (May), 439–458.

KENNEDY, K. AND KREMER, U. 1995. Automatic data layout for High Performance Fortran. In *Proceedings of Supercomputing '95*. IEEE Computer Society Press, Los Alamitos, CA.

KICZALES, G. 1986. Beyond the black box: open implementation. *IEEE Software 13,* 1 (Jan.).

KIM, J.-H. AND VAIDYA, N. H. 1996. A cost-comparison approach for adaptive distributed shared memory. In *Proceedings of the 1996 ACM International Conference on Supercomputing.* ACM Press, New York, NY.

KNUTH, D. 1971. An empirical study of FORTRAN programs. *Software—Practice and Experience 1,* 105–133.

LEE, P. AND LEONE, M. 1996. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN '96 Conference on Program Language Design and Implementation.* ACM Press, New York, NY, 137–148.

LENOSKI, D. 1992. The design and analysis of DASH: A scalable directory-based multiprocessor. Ph.D. thesis, Stanford, CA.

LEUNG, S. AND ZAHORJAN, J. 1993. Improving the performance of runtime parallelization. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM Press, New York, NY, 83–91.

LIM, B.-H. AND AGARWAL, A. 1994. Reactive synchronization algorithms for multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, New York, NY, 25–37.

MOON, S., HALL, M., AND MURPHY, B. 1998. Predicated array data-flow analysis for run-time parallelization. In *Proceedings of the 1998 ACM International Conference on Supercomputing.* ACM Press, New York, NY, 204–211.

MORRIS, W. 1991. CCG: a prototype coagulating code generator. In *Proceedings of the ACM SIGPLAN '91 Conference on Program Language Design and Implementation*. ACM Press, New York, NY, 45–58.

PETTIS, K. AND HANSEN, D. 1990. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Program Language Design and Implementation*. ACM Press, New York, NY, 16–27.

PLEVYAK, J., ZHANG, X., AND CHIEN, A. 1995. Obtaining sequential efficiency for concurrent object-oriented languages. In *Proceedings of the Twenty-second Annual ACM Symposium on the Principles of Programming Languages*. ACM Press, New York, NY, 311–321.

POLYCHRONOPOULOS, C. AND KUCK, D. 1987. Guided self-scheduling: A practical scheduling scheme for parallel computers. *IEEE Transactions on Computers 36,* 12 (Dec.), 1425–1439.

RAUCHWERGER, L. AND PADUA, D. 1995. The LRPD test: speculative run-time parallelization of loop with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN '95 Conference on Program Language Design and Implementation.* ACM Press, New York, NY, 218–232.

RINARD, M. AND DINIZ, P. 1997. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems 19,* 6 (Nov.), 941–992.

RINARD, M., SCALES, D., AND LAM, M. 1992. Heterogeneous Parallel Programming in Jade. In *Proceedings of Supercomputing '92*. IEEE Computer Society Press, Los Alamitos, CA, 245–256.

ROMER, T., LEE, D., B.BERSHAD, AND CHEN, J. 1994. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*. ACM Press, New York, NY, 255 –266.

SAAVEDRA, R. AND PARK, D. 1996. Improving the effectiveness of software prefetching with adaptive execution. In *Proceedings of the Parallel Architectures and Compilation Techniques (PACT'96)*. IEEE Computer Society Press, Los Alamitos, CA.

SALTZ, J., BERRYMAN, H., AND WU, J. 1991. Multiprocessors and run-time compilation. *Concurrency: Practice & Experience 3,* 6 (Dec.), 573–592.

SINGH, J., WEBER, W., AND GUPTA, A. 1992. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News 20,* 1 (March), 5–44.

SMITH, M. 1991. Tracing with Pixie. Tech. Rep. CSL-TR-91-497, Stanford University. November.

WALL, D. 1986. Global register allocation at link time. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.* ACM Press, New York, NY, 264–275.

ZHANG, X., WANG, Z., GLOY, N., CHEN, J. B., AND SMITH, M. D. 1997. System support for automatic profiling and optimization. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. ACM Press, New York, NY, 15–25.