Dynamic Feedback: An Effective Technique for Adaptive Computing *

Pedro Diniz and Martin Rinard
Department of Computer Science
Engineering I Building
University of California, Santa Barbara
Santa Barbara, CA 93106-5110
{pedro,martin}@cs.ucsb.edu

Abstract

This paper 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.

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 [12]. The best data distribution of dense matrices in distributed memory machines depends on how the different parts of the program access the matrices [1, 2, 18, 21]. The best concrete data structure to implement a given abstract data type often depends on how it is used [14, 22]. The best algorithm to solve a given problem often depends on the combination of input and hardware platform used to execute the algorithm [5]. 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 new 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. The extreme heterogeneity of such systems will defeat any implementation that does not adapt to different execution environments. Dynamic feedback is one example of the adaptive techniques that will enable compilers to deliver good performance in modern computing systems.

This paper describes the use of dynamic feedback 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 [33]. Our experimental results show that the resulting synchronization overhead can significantly degrade the performance [10]. We have developed a set of synchronization transformations and a set of synchronization optimization policies that use the transformations to reduce the synchronization overhead [10]. 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 infor-

*Pedro Diniz is sponsored by the PRAXIS XXI program administrated by JNICT – Junta Nacional de Investigaçao Científica e Tecnológica from Portugal, and holds a Fulbright travel grant. Martin Rinard is supported in part by an Alfred P. Sloan Research Fellowship.

mation, 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 paper 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 the generated code exhibits performance comparable to that of code that has been manually tuned to use the best synchronization optimization policy.

1.2 Structure

The remainder of the paper is structured as follows. In Section 2, we briefly summarize the analysis technique, commutativity analysis, that our compiler is based on. Section 3 summarizes the issues that affect the performance impact of the synchronization optimization algorithms. Section 4 describes the implementation details of applying dynamic feedback to the problem of choosing the best synchronization policy. Section 5 presents the theoretical analysis. Section 6 presents the experimental results. We discuss related work in Section 7 and conclude in Section 8.

2 Commutativity Analysis

Our compiler uses *commutativity analysis* to automatically parallelize serial, object-based programs. Such programs structure the computation as a sequence of *operations* on *objects*. The compiler analyzes the program at this granularity to determine when operations commute, or generate the same result regardless of the order in which they execute. If all of the operations in a given computation commute, the compiler can automatically generate parallel code. This code executes all of the operations in the computation in parallel. Our experimental results indicate that this approach can effectively parallelize irregular computations that manipulate dynamic, linked data structures such as trees and graphs [33].

To ensure that operations 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.

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 [10]. 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 [10]. The result is a computation that acquires and releases the lock only once. In effect, the 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 1 and 2 present an example of how synchronization optimizations can reduce the number of executed acquire and release constructs. Figure 1 presents a program (inspired by the Barnes-Hut benchmark described in Section 6) that uses mutual exclusion locks to make `body::one_interaction` operations execute atomically. Figure 2 presents the program after the application of a synchronization optimization algorithm. The 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:

```
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]);
  }
}
```

Figure 1: 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();
}
```

Figure 2: Optimized Example Computation

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

To obtain the optimality results in Section 5, the generated code for the sampling phase must execute each policy for a fixed sampling time interval. The production phase must also execute 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. 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.

- **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 9 microseconds, which is negligible compared with the sizes of the iterations of the parallel loops in our benchmark 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.[1]

  One potential drawback of synchronous switching is that 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.

- **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 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. The theoretical analysis in Section 5 formulates some of the optimality results in terms of the effective sampling 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 1 presents the sizes of the text segments for several different versions of our benchmark applications. 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 1: Executable Code Sizes (bytes)

We also considered using dynamic compilation [3, 11, 24] 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 sampling phases could be significantly longer than our set of benchmark applications would tolerate.

Finally, it is possible for the compiler to generate a single 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

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

[1] 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.

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. Our experimental results indicate that this overhead has little or no effect on the performance. We measure the overhead by generating versions of the applications that use a single, statically chosen, synchronization optimization policy. We then execute these versions with the instrumentation turned on and the instrumentation turned off. The performance differences between the instrumented and uninstrumented versions are very small, which indicates that the instrumentation overhead has 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. Section 5 presents a theoretical analysis that characterizes how long the production phase must be relative to the sampling phase to achieve an optimality result. 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 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 with 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.

5 Theoretical Analysis

In this section, we present a theoretical analysis of the worst-case performance of dynamic feedback. We compare dynamic feedback with an hypothetical, unrealizable algorithm that always uses the best policy.

We start by observing that if there is no constraint on how fast the overhead of each policy may change, it is impossible to obtain a meaningful optimality result for any sampling algorithm — the overhead of each policy may change dramatically right after the sampling phase. We therefore impose the constraint that changes in the overheads of the different policies are bounded by an exponential decay function. We also assume that the values measured during the sampling phase accurately reflect the actual overheads at the start of the production phase. Finally, we assume each production phase executes to completion, although it is possible to relax this assumption.

The worst case for the dynamic feedback algorithm relative to the optimal algorithm occurs when more than one policy has the lowest overhead during the sampling phase. In this case, the dynamic feedback algorithm must arbitrarily select one of the sampled policies with the lowest overhead for the production phase. The maximum difference between the performance of the dynamic feedback algorithm and the performance of the optimal algorithm occurs when the overhead of the selected policy increases at the maximum bounded rate and the overheads of the other policies decrease at the maximum bounded rate. We analyze this scenario to derive a conservative bound on the worst-case performance of the dynamic feedback algorithm relative to the optimal algorithm.

We next establish some notation. The variable S is the effective sampling interval, P is the length of the production interval and $p_0, \cdots, p_{N-1}$ are the N different policies. The computation starts with a sampling phase. During this phase, the dynamic feedback algorithm executes each of the N policies for the sampling interval S to derive overhead measurements $v_0, \cdots, v_{N-1}$ for each of the N policies. The overhead is the proportion of the total execution time spent executing lock constructs or waiting for other processors to release locks. The overhead therefore varies between zero (if the computation never executes a lock construct) and one (if the computation performs no useful work). In the worst case, multiple policies have the same overhead v during the sampling phase, and v is the lowest sampled overhead.

Without loss of generality, we assume that the dynamic feedback algorithm executes policy p_0 during the production interval. We also assume that, at any time t during the production phase, the policy overheads are bounded above by the exponential decay function $1+(v-1)e^{-\lambda t}$ (here λ is the rate of decay). In the worst case, the overhead function $o_0(t)$ of policy p_0 actually hits this bound:

$$o_0(t) = 1 + (v-1)e^{-\lambda t} \tag{1}$$

We define the amount of useful work performed by a given policy p_i over given period of time T by:

$$\mathtt{Work}_i^T = \int_0^T (1 - o_i(t))dt \tag{2}$$

The dynamic feedback algorithm performs the following amount of work during the production phase:

$$\mathtt{Work}_0^P = \frac{(1-v)}{\lambda}(1 - e^{-\lambda P}) \tag{3}$$

Our worst-case analysis conservatively assumes that no useful work at all takes place during the sampling phase. For the dynamic feedback algorithm, $\mathtt{Work}_0^P$ is therefore the total amount of useful work performed during the $SN + P$ units of time that make up the sampling and production phases.

We now turn our attention to the optimal algorithm. When it begins executing, multiple policies have the same lowest overhead v. We assume that, at any time t during the first P time units, the policy overheads are bounded below by the exponential decay function $ve^{-\lambda t}$.

In the worst case, the overhead function of one of the policies, say policy p_1, actually hits the bound:

$$o_1(t) = ve^{-\lambda t} \tag{4}$$

In this case the optimal algorithm will execute policy p_1 for the first P time units, and the total useful work performed during this interval is:

$$\mathtt{Work}_1^P = P - \frac{v}{\lambda}(1 - e^{-\lambda P}) \tag{5}$$

We make the conservative assumption that, for the next SN time units, the optimal algorithm executes a policy with no overhead at all — in other words, that it performs SN units of work during this time period.

We now compare the amounts of work performed by the worst-case dynamic feedback algorithm and the best case optimal algorithm over the time period $P + SN$:

$$\mathtt{Work}_1^{P+SN} - \mathtt{Work}_0^{P+SN} = SN + P + \frac{1}{\lambda}e^{-\lambda P} - \frac{1}{\lambda} \tag{6}$$

We next discuss the conditions under which we can obtain a guaranteed performance bound for the dynamic feedback algorithm relative to the optimal algorithm. We start by defining a precise way to compare policies:

Definition 1 *Policy p_i is at most ϵ worse than policy p_j over a time interval T if $\mathtt{Work}_i^T - \mathtt{Work}_j^T \leq T\epsilon$.*

Given a decay rate λ, an effective sampling interval S, a number of policies N and a desired performance bound ϵ, Definition 1 yields the following inequality, which determines if it is possible to choose a production interval P such that the dynamic feedback algorithm is guaranteed to be most ϵ worse than the optimal algorithm. If so, the inequality also characterizes the values of P that are guaranteed to deliver the desired performance.

$$(1-\epsilon)P + \frac{1}{\lambda}e^{-\lambda P} \leq (\epsilon - 1)SN + \frac{1}{\lambda} \tag{7}$$

Conceptually, several things are happening in this inequality. First, the production interval P must be long enough to successfully amortize the sampling time SN. Second, the production interval P must be short enough so that the dynamic feedback algorithm detects policy overhead changes quickly enough to avoid executing an inefficient policy for a long time. In other words, the inequality bounds P both below and above. Finally, the decay rate λ must be small enough so that the dynamic feedback algorithm can perform enough work relative to the optimal algorithm to obtain the bound.

In some cases, it is impossible to choose a P that satisfies the conditions. If it is possible to choose such a P, the inequality identifies a *feasible region* in which P is guaranteed to satisfy the conditions. Figure 3 graphically illustrates the range of feasible values for the production interval P using the following example values: $S = 1.0$, $N = 2$, $\lambda = 0.065$ and $\epsilon = 0.5$. The inequality also provides insight into various relationships. As ϵ increases, the range of feasible values for P also increases. As S increases, the range of feasible values for P decreases.

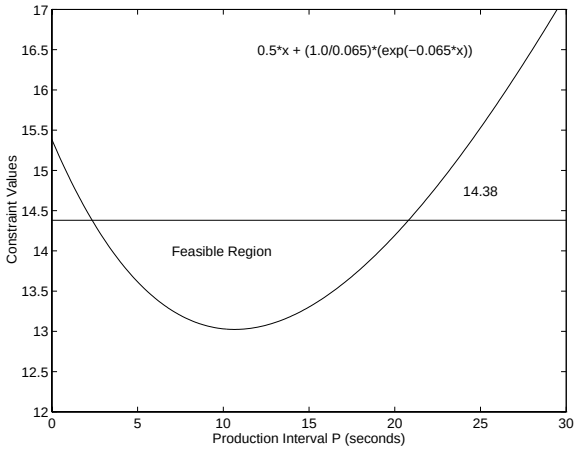

Figure 3: Feasible Region for Production Interval P

We next show how to determine the optimal value of P under our worst-case assumptions. We define the optimal value for P as the value that minimizes the worst-case difference in work performed per unit time by the optimal and dynamic feedback algorithms. Equation 8 defines this difference.

$$\frac{\mathtt{Work}_1^{P+SN} - \mathtt{Work}_0^{P+SN}}{P + SN} = \frac{SN + P + \frac{1}{\lambda}e^{-\lambda P} - \frac{1}{\lambda}}{P + SN} \tag{8}$$

Finding the root of the first derivative and solving for P yields the following equation. The value P_{opt} that satisfies this equation

is the optimal value of P. It is possible to use numerical methods to solve for P_{opt}.

$$e^{-\lambda P}(P + SN + \frac{1}{\lambda}) = \frac{1}{\lambda} \quad (9)$$

For the example values used in Figure 3, the optimal value of P is $P_{opt} \approx 7.25$.

6 Experimental Results

This section presents experimental results that characterize how well dynamic feedback works for three benchmark applications. The applications are Barnes-Hut [4], a hierarchical N-body solver, Water [38], which simulates water molecules in the liquid state, and String [19], which builds a velocity model of the geology between two oil wells. 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 [25] 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.

6.1 Barnes-Hut

Table 2 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.[2]

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 2: Execution Times for Barnes-Hut (seconds)

[2] Strictly speaking, the Dynamic version only needs to execute instrumented code 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.

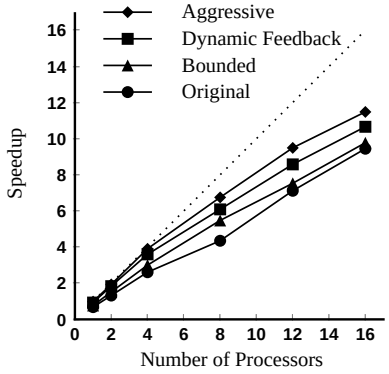

Figure 4: Speedups for Barnes-Hut

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.

Table 3 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 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 3: 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 [33].

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.

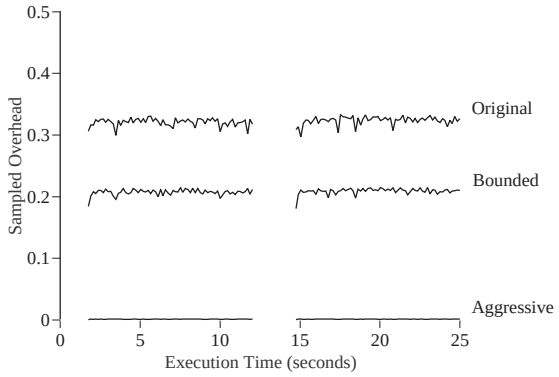

Figure 5: Sampled Overhead for the Barnes-Hut FORCES Section on Eight Processors

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 4 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 of useful work in the section. Because the generated code checks for the expiration of sampling and production intervals only at the granularity of the loop iterations, the sizes of the loop iterations have an important impact on the size of the minimum effective sampling interval.

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

Table 4: 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 5 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.

We next consider the impact of varying the target sampling and production intervals. For the performance numbers in Table 2, the target sampling interval was set to ten 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 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.

Version	Mean Minimum Effective Sampling Interval (milliseconds)
Original	10
Bounded	7.8
Aggressive	6.5

Table 5: Mean Minimum Effective Sampling Intervals for the Barnes-Hut FORCES Section on Eight Processors

Table 6 presents the mean execution times of the FORCES section running on eight processors for several combinations of target sampling and production intervals. The performance is relatively insensitive to the variation in the target sampling and production intervals. Even when the target sampling and production intervals are identical (which means the computation spends approximately three times as long in the sampling phase as in the production phase), the section only runs approximately 20% slower than with the best combination.

Target Sampling Interval	Target Production Interval			
	1 second	5 seconds	10 seconds	1000 seconds
0.01 seconds	9.138	9.058	9.058	9.025
0.1 seconds	9.697	9.178	9.122	9.220
1.0 seconds	10.784	9.834	9.726	9.670

Table 6: Mean Execution Times for Varying Production and Sampling Intervals for the Barnes-Hut FORCES Section on Eight Processors (seconds)

6.2 Water

Table 7 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 7: 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. As the performance results presented

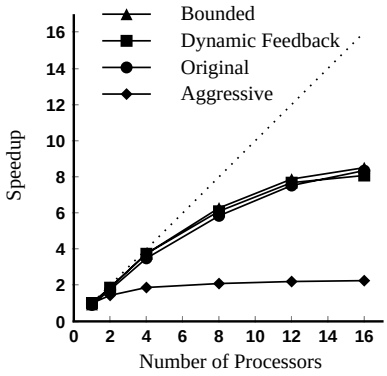

Figure 6: Speedups for Water

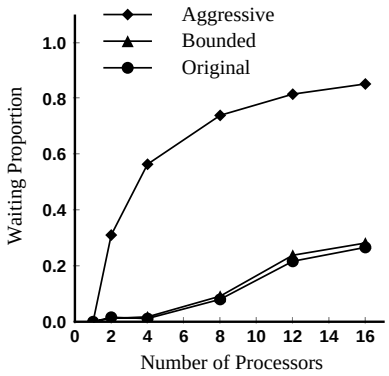

Figure 7: Waiting Proportion for Water

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 8 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 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 8: 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.[3] These data were collected using program-counter sampling to profile the execution [16, 23]. 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

[3] 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.

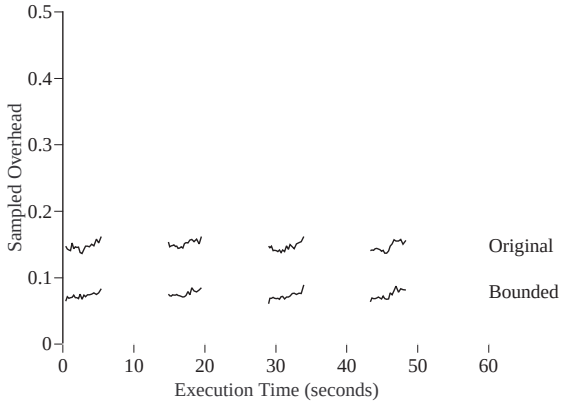

Figure 8: Sampled Overhead for the Water INTERF Section on Eight Processors

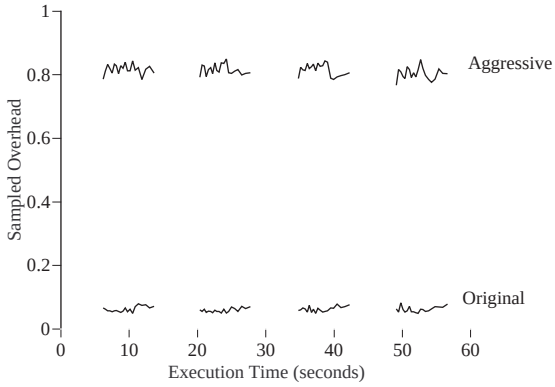

Figure 9: Sampled Overhead for the Water POTENG Section on Eight Processors

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 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.

Tables 9 and 10 present the parallel section statistics for the INTERF and POTENG sections. Tables 11 and 12 present the mean minimum effective sampling intervals 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.

Mean Section Size	Number of Iterations	Mean Iteration Size
20.80 seconds	511	40.7 milliseconds

Table 9: Statistics for the Water INTERF Section

Mean Section Size	Number of Iterations	Mean Iteration Size
16.34 seconds	511	32.0 milliseconds

Table 10: Statistics for the Water POTENG Section

Version	Mean Minimum Effective Sampling Interval (milliseconds)
Original	93
Bounded	82

Table 11: Mean Minimum Effective Sampling Intervals for the Water INTERF Section on Eight Processors

Version	Mean Minimum Effective Sampling Interval (milliseconds)
Original	59
Aggressive	286

Table 12: Mean Minimum Effective Sampling Intervals for the Water POTENG Section

For the performance numbers in Table 7, the target sampling interval was set to ten 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. Tables 13 and 14 present the execution times for the INTERF and POTENG sections 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.

For target production intervals of one and five seconds, the performance of the POTENG section is quite sensitive to the choice of target sampling interval. 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

Target Sampling Interval	Target Production Interval			
	1 second	5 seconds	10 seconds	1000 seconds
0.01 seconds	3.669	3.541	3.507	3.552
0.1 seconds	3.733	3.532	3.529	3.566
1.0 seconds	3.713	3.695	3.676	3.673

Table 13: Mean Execution Times for Varying Production and Sampling Intervals for the Water INTERF Section on Eight Processors (seconds)

Target Sampling Interval	Target Production Interval			
	1 second	5 seconds	10 seconds	1000 seconds
0.01 seconds	3.32	2.624	2.642	2.649
0.1 seconds	3.072	2.690	2.709	2.724
1.0 seconds	4.184	3.482	3.479	3.489

Table 14: Mean Execution Times for Varying Production and Sampling Intervals for the Water POTENG Section on Eight Processors (seconds)

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 five, ten and 1000 seconds. We attribute this uniformity to the fact that the execution of the POTENG section always terminates in less than five seconds. Extending the target production interval beyond five seconds therefore has no effect on the execution.

Second, the execution times are virtually identical for a target production interval of 5.0 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 five 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 1.0 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 increase in the target sampling interval is a significant reduction in the amount of time spent executing the inefficient Aggressive sampling intervals.

6.3 String

For String, the Bounded policy produces the same parallel code as the Original policy. We therefore report performance results for only the Original, Aggressive and Dynamic policies. Table 15 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.

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 15: Execution Times for String (seconds)

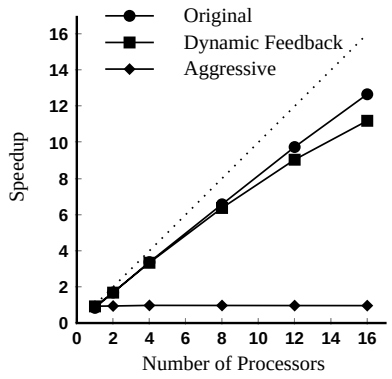

Figure 10: Speedups for String

For String, the Aggressive policy completely serializes the computation. This 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 16 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 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.

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

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 16: Locking Overhead for String

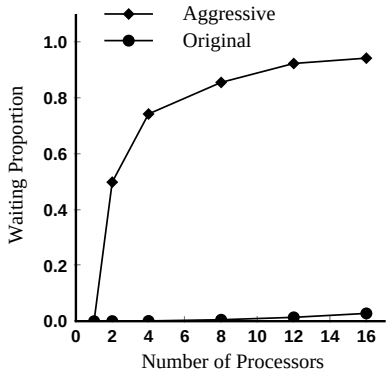

Figure 11: Waiting Proportion for String

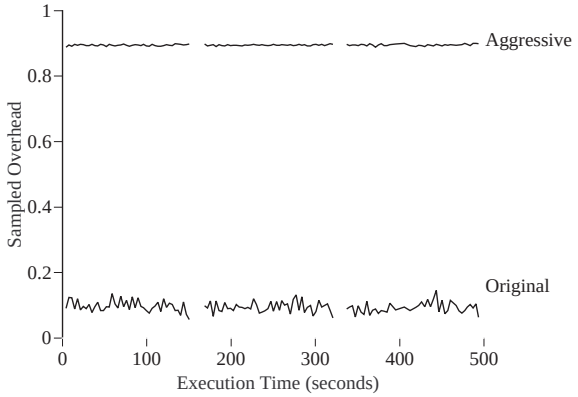

Figure 12: Sampled Overhead for the String PROJFWD Section on Eight Processors

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

Table 17: Statistics for the String PROJFWD Section

Version	Mean Minimum Effective Sampling Interval
Original	54 milliseconds
Aggressive	260 milliseconds

Table 18: Mean Minimum Effective Sampling Intervals for the String PROJFWD Section

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 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 17 presents the parallel section statistics for the PROJFWD section. Table 18 presents the mean minimum effective sampling intervals. 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 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 15, the target sampling interval was set to ten 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 19 presents the execution times for the PROJFWD section running on eight processors for several combinations of target sampling and production intervals. As expected for 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.

Target Sampling Interval	Target Production Interval			
	1 second	5 seconds	10 seconds	1000 seconds
0.01 seconds	140.6	117.1	114.7	112.54
0.1 seconds	144.7	118.3	114.3	112.69
1.0 seconds	165.5	131.1	121.7	112.96

Table 19: Mean Execution Times for Varying Production and Sampling Intervals for the String PROJFWD Section on Eight Processors (seconds)

6.4 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 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, dynamic type feedback techniques for improving the performance of object-oriented languages, adaptive execution techniques and dynamic techniques for parallelizing loops. We also discuss dynamic compilation, efficient implementations of parallel function calls, and related work in synchronization optimization.

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.

Profiling has been used in the context of object-oriented languages to predict the most frequently occurring class of the receiver object at a given call site [17]. This information is then used to drive optimizations that inline methods based on predictions about the class of the receiver. Profiling has also been used to guide decisions to inline procedures in C programs [7], to drive instruction scheduling algorithms [8], to help place code so as to minimize the impact on the memory hierarchy [29], to aid in register allocation [28, 39], and to direct the compiler to frequently executed parts of the program so that the compiler can apply further optimizations [13].

Brewer [5] 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.

Dynamic feedback differs from profile-based feedback in that it can adapt dynamically to the current execution environment, rather than hoping that the environment is similar to the environment in the profiling run of the program. Dynamic feedback can also adjust to changes that occur within a single execution. Profile-based approaches collect a single aggregate set of measurements for the entire execution, and can therefore miss environment changes that take place within a single execution.

7.2 Adaptive Execution Techniques

Other researchers have recognized the need to use dynamic performance data to optimize the execution [9, 35, 36]. These approaches are based on a set of control variables that parameterize a given algorithm in the implementation. An example of a control variable is the prefetch distance in an algorithm that prefetches data accessed by a loop [36]. Typically, the programmer defines the set of observable variables and a feedback function that uses the observable variables to produce 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.

While dynamic feedback is similar in spirit to these approaches, there is an important difference. Dynamic feedback is a general technique designed to choose between several discrete, and potentially quite different, implementations. Other approaches are designed to tune one or more control variables in the context of a single algorithm.

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 [6].

Dynamic type feedback is designed to direct the compiler's attention to parts of the program that would benefit from optimization [20]. 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 [26, 37]. The Jade implementation dynamically analyzes how tasks access data to exploit the concurrency in coarse-grain parallel programs [34]. Speculative approaches optimistically execute loops in parallel, rolling back the computation if the parallel execution violates the data dependences [32].

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 [3, 11, 24]. 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 paper applies dynamic feedback to the problem of choosing the best synchronization granularity. Our previous research produced analyses and transformations for reducing the synchronization overhead and the different synchronization optimization policies [10]. Plevyak, Zhang and Chien have developed a similar synchronization optimization technique, *access region expansion*, for concurrent object-oriented programs [31]. 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.

7.7 Parallel Function Calls

Several researchers have developed efficient implementations for parallel function calls [15, 27, 30]. These implementations dynamically match the amount of exploited parallelism to the amount of parallelism available on the parallel hardware platform by selecting between an efficient sequential call and a full parallel call. The selection is based on a dynamic measure of the difference between the currently exploited and available amounts of parallelism.

8 Conclusion

This paper presents a new 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. 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.

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. The extreme heterogeneity of such systems will defeat any implementation that does not adapt to different execution environments. Dynamic feedback is one example of the adaptive techniques that will enable compilers to deliver good performance in modern computing systems.

References

[1] S.P. Amarasinghe and M.S. Lam. Communication optimization and code generation for distributed-memory machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, SIGPLAN Notices 28(7). ACM, July 1993.

[2] J.M. Anderson and M.S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, SIGPLAN Notices 28(7). ACM, July 1993.

[3] J. Auslander, M. Philipose, C. Chambers, S. Eggers, , and B. Bershad. Fast, effective dynamic compilation. In *Proceedings of the SIGPLAN '96 Conference on Program Language Design and Implementation*, Philadelphia, PA, May 1996.

[4] J. Barnes and P. Hut. A hierarchical O(NlogN) force-calculation algorithm. *Nature*, pages 446–449, December 1976.

[5] E. Brewer. High-level optimization via automated statistical modeling. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[6] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[7] P. Chang, S. Mahlke, W. Chen, and W. Hwu. Profile-guided automatic inline expansion for C programs. *Software—Practice and Experience*, 22(5):349–369, May 1992.

[8] W. Chen, S. Mahlke, N. Warter, S. Anik, and W. Hwu. Profile-assisted instruction scheduling. *International Journal of Parallel Programming*, 22(2):151–181, April 1994.

[9] A. Cox and R. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993.

[10] P. Diniz and M. Rinard. Synchronization transformations for parallel computing. In *Proceedings of the Twenty-fourth Annual ACM Symposium on the Principles of Programming Languages*, Paris, France, January 1997. ACM.

[11] D. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the SIGPLAN '96 Conference on Program Language Design and Implementation*, Philadelphia, PA, May 1996.

[12] B. Falsafi, A. Lebeck, S. Reinhardt, I. Schoinas, M. Hill, J. Larus, A. Rogers, and D. Wood. Application-specific protocols for user-level shared memory. In *Proceedings of Supercomputing '94*, November 1994.

[13] M. Fernandez. Simple and effective link-time optimization of modula-3 programs. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995.

[14] S. Freudenberger, J. Schwartz, and M. Sharir. Experience with the SETL optimizer. *ACM Transactions on Programming Languages and Systems*, 5(1):26–45, January 1983.

[15] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy Threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, August 1996.

[16] S. Graham, P. Kessler, and M. McKusick. gprof: a call graph execution profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, Boston, MA, June 1982.

[17] D. Grove, J. Dean, C. Garret, and C. Chambers. Profile-guided receiver class prediction. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Austin, TX, October 1995.

[18] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[19] J. Harris, S. Lazaratos, and R. Michelena. Tomographic string inversion. In *60th Annual International Meeting, Society of Exploration and Geophysics, Extended Abstracts*, pages 82–85, 1990.

[20] U. Holzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*, Orlando, FL, June 1994.

[21] K. Kennedy and U. Kremer. Automatic data layout for High Performance Fortran. In *Proceedings of Supercomputing '95*, San Diego, CA, November 1995.

[22] G. Kiczales. Beyond the black box: open implementation. *IEEE Software*, 13(1), January 1986.

[23] D. Knuth. An empirical study of FORTRAN programs. *Software—Practice and Experience*, 1:105–133, 1971.

[24] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *Proceedings of the SIGPLAN '96 Conference on Program Language Design and Implementation*, Philadelphia, PA, May 1996.

[25] D. Lenoski. *The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor*. PhD thesis, Stanford, CA, February 1992.

[26] S. Leung and J. Zahorjan. Improving the performance of runtime parallelization. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–91, San Diego, CA, May 1993.

[27] E. Mohr, D. Kranz, and R. Halstead. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 185–197, June 1990.

[28] W. Morris. Ccg: a prototype coagulating code generator. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[29] K. Pettis and D. Hansen. Profile guided code positioning. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, White Plains, NY, June 1990.

[30] J. Plevyak, V. Karamcheti, X. Zhang, and A. Chien. A hybrid execution model for fine-grained languages on distributed memory multicomputers. In *Proceedings of Supercomputing '95*, San Diego, CA, November 1995.

[31] J. Plevyak, X. Zhang, and A. Chien. Obtaining sequential efficiency for concurrent object-oriented languages. In *Proceedings of the Twenty-second Annual ACM Symposium on the Principles of Programming Languages*. ACM, January 1995.

[32] L. Rauchwerger and D. Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995.

[33] M. Rinard and P. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Proceedings of the SIGPLAN '96 Conference on Program Language Design and Implementation*, Philadelphia, PA, May 1996. (http://www.cs.ucsb.edu/∼martin/paper/pldi96.ps).

[34] M. Rinard, D. Scales, and M. Lam. Heterogeneous Parallel Programming in Jade. In *Proceedings of Supercomputing '92*, pages 245–256, November 1992.

[35] T. Romer, D. Lee, B.Bershad, and J. Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 255 –266, Monterey, CA, November 1994.

[36] R. Saavedra and D. Park. Improving the effectiveness of software prefetching with adaptive execution. In *Proceedings of the 1996 Conference on Parallel Algorithms and Compilation Techniques (PACT '96*, Boston, MA, October 1996.

[37] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and run-time compilation. *Concurrency: Practice & Experience*, 3(6):573–592, December 1991.

[38] J. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[39] D. Wall. Global register allocation at link time. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*. ACM, June 1986.