Parallel Application Characterization for Multiprocessor Scheduling Policy Design

Thu D. Nguyen, Raj Vaswani, and John Zahorjan

Department of Computer Science and Engineering, Box 352350
University of Washington, Seattle, WA 98195-2350 USA

Abstract. Much of the recent work on multiprocessor scheduling disciplines has used abstract workload models to explore the fundamental, high-level properties of the various alternatives. As continuing work on these policies increases their level of sophistication, however, it is clear that the choice of appropriate policies must be guided at least in part by the typical behavior of actual parallel applications. Our goal in this paper is to examine a variety of such applications, providing measurements of properties relevant to scheduling policy design. We give measurements for both hand-coded parallel programs (from the SPLASH benchmark suites) and compiler-parallelized programs (from the PERFECT Club suite) running on a KSR-2 shared-memory multiprocessor.

The measurements we present are intended primarily to address two aspects of multiprocessor scheduling policy design:

- In the spectrum between aggressively dynamic and static allocation policies, what is an appropriate choice for the rate at which reallocations should take place?
- Is it possible to take measurements of application speedup and efficiency at runtime that are sufficiently accurate to guide allocation decisions?

We address these questions through three sets of measurements:

- First, we examine application speedup, and the sources of speedup loss. Our results confirm that there is considerable variation in job speedup, and that the bulk of the speedup loss is due to communication and idleness.
- Next, we examine runtime measurement of speedup information. We begin by looking at how such information might be acquired accurately and at acceptable cost. We then investigate the extent to which recent measurements of speedup accurately predict the future, and so the extent to which such measurements might reasonably be expected to guide allocation decisions.
- Finally, we examine the durations of individual processor idle periods, and relate these to the cost of reallocating a processor at those times. These results shed light on the potential for aggressively dynamic policies to improve performance.

1 Introduction

A quantitative understanding of realistic workload characteristics is critical to the design of processor scheduling policies. Because such information has not been widely

This work was supported in part by the National Science Foundation (Grants CCR-9123308 and CCR-9200832) and the Washington Technology Center.

available, many scheduling studies have been performed using analytic or synthetic workload models [19, 17, 29, 5]. While such artificial workloads are a valuable tool, the increasing sophistication of the policies being studied requires a corresponding increase in the sophistication of the workload models. Our purpose in this paper is to identify and quantify workload characteristics that will help formulate and parameterize such models.

We address this issue in the context of multiprogrammed shared-memory multiprocessors through measurements of seventeen scientific applications when run on a 60 node KSR-2. The measurements we present are intended primarily to address two aspects of multiprocessor scheduling policy design:

- In the spectrum between aggressively dynamic and static allocation policies, what is an appropriate choice for the rate at which reallocations should take place?
- Is it possible to take measurements of application speedup and efficiency at runtime that are sufficiently accurate to guide allocation decisions?

We address these questions through three sets of measurements. First, we examine application speedup, and the sources of speedup loss. Our results confirm that there is considerable variation among jobs, and provide information that will support work on the use of speedup information in making scheduling decisions [13, 23].

Second, because it is at least burdensome, and perhaps impossible, to accurately collect and supply such information at job submission time, we look at the problem of estimating job speedup at runtime. We first demonstrate a technique for estimating "instantaneous speedup" at runtime that is both efficient and accurate. We next examine the extent to which recent measurements of application speedup accurately predict the near future. Clearly, for runtime measurements to be useful in scheduling, such predictions must be reliable. We find that this is a difficult problem, and propose what is at least a first step solution that engages the cooperation of the application in making measurements.

Finally, observing that idleness is a significant source of speedup loss, we examine the durations of individual processor idle periods, and relate these to the cost of reallocating a processor at those times. We find that while most idle periods are too short to justify reallocating a processor, there is still considerable idleness in those fewer periods of greater length. We examine how much processor time could be recovered by reallocation based on various conservative assumptions about the cost of doing so.

Because we believe that realistic workloads will be comprised of applications that are implemented using a variety of methodologies, our application suite consists of programs from two distinct development domains: (1) hand-coded parallel applications that implement sophisticated parallel algorithms and may include optimizations such as load balancing and careful data partitioning to minimize communication, and (2) compiler-parallelized sequential programs.

We use programs from the SPLASH and SPLASH-2 benchmark suites [30, 37] to represent hand-coded parallel applications, and programs from the PERFECT Club benchmark suite [3] and an industrial fluid dynamics program (obtained from Analytical Methods, Inc.) to represent compiler-parallelized sequential applications.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 documents our experimental platform. Section 4 looks at application speedup,

and identifies and quantifies the major components of speedup loss. Section 5 examines the question of whether instantaneous speedup varies significantly during execution, and so sheds light on the extent to which recent measurements of application performance predict its future behavior. Section 6 reports measurements of the frequency and duration of processor idleness, characteristics important in deciding how aggressive schedulers should be in reallocating idle processors. Section 7 gives our conclusions.

2 Related Work

The PERFECT Club benchmark suite is one of several standard benchmark suites used to measure the capability of parallelizing compilers [3]. As such, many studies have characterized the behavior of a number of these PERFECT Club programs, e.g., [7, 11, 26]. However, these studies have typically focused on properties of the code that affect a compiler's ability to parallelize them, and not, as we do, on properties of their execution that affect a scheduler's ability to best schedule parallelized versions of the programs.

For the SPLASH and SPLASH-2 benchmark suites, Singh et al. [30] and Woo et al. [37] provide significant information, including speedup, cache behavior, and synchronization wait time. Our measurements supplement their reports by quantifying sources of speedup loss, as well as more fine-grained application behaviors such as frequency and duration of idle periods. Furthermore, we contrast the behaviors of these applications with those of compiler-parallelized applications, and consider the implications of their differences to the design of parallel processor scheduling policies.

Feitelson and Nitzberg [12] report a variety of statistics on the parallel workloads of an iPSC/860 located at NASA Ames. They discuss what we call submitted workload mix characteristics, such as the ratio of sequential to parallel jobs, resource usage patterns (e.g., the correlation between total resource requirement and the degree of parallelism), job submission rates, and system utilization. In contrast, we characterize fine-grained behaviors of individual applications in order to answer questions such as "*how often do applications idle one or more of their allocated processors?*" and "*are idle periods long enough with respect to reallocation cost such that reallocation in response to application idleness can improve system utilization?*".

Cypher et al. [8] report measurements for a number of applications running on message-passing multiprocessor systems in a manner similar to ours. However, because they were attempting to address architectural issues, they concentrate on different measures (e.g., memory and I/O requirements) than those presented here. Similarly, many other researchers (e.g., [9, 1, 27]) report results from studies of application memory behavior.

3 The Experimental Environment

3.1 Hardware and Software Platform

All measurements were done on a Kendall Square Research KSR-2 COMA shared-memory multiprocessor. Our machine consists of 60 40-MHz dual-issue proprietary

processors, partitioned into two clusters of 30. Each processor is connected to a 256-KByte data cache, 256-KByte instruction cache, and a 32-MByte attraction memory. Processors in each cluster, and the clusters themselves, are connected by separate 1 GB/s. slotted ring networks[1]. The attraction memories cooperate to implement a sequentially consistent, globally-shared address space. The unit of transfer and sharing between attraction memories is 128 bytes.

Each node in the KSR-2 contains a hardware monitoring unit called the *Event Monitor* that compiles information such as cache misses and processor stall (communication) time. This information is made available to the system and user jobs through a set of read-only registers.

The KSR-2 runs a variant of the OSF/1 UNIX operating system. We use CThreads [6], an efficient user-level threads package, as the vehicle of parallelism. We instrumented CThreads using the event monitors to collect the data presented in the remainder of this paper.

SPLASH and SPLASH-2 programs run directly on CThreads. We use both the KSR KAP [16] and Stanford SUIF compilers [36] to parallelize sequential programs. We use both systems because they represent different tradeoffs in technology and product maturity. KSR KAP is a commercial product that has been adapted specifically to the KSR architecture and optimized through productization. SUIF, on the other hand, is a research vehicle. As such, SUIF implements many state-of-the-art parallelization techniques not present in KAP, but has been less concerned with standard optimizations and has not been tuned to the KSR architecture.

3.2 Applications

Tables 1 and 2 list the applications that we measured, and give brief descriptions of each as well as their execution time when run on a single processor[2]. This single-processor time represents the execution of a parallel version of the program running on a single processor, not of a sequential version of the program. We use these as the base times in computing speedup, rather than the times for true sequential versions, because our interest is in schedulers; the performance gap between the sequential version and parallel version executing on a single processor highlights the weakness of the program or compiler, but does not present an opportunity exploitable by the scheduler.

For reasons of space, in what follows, we show results for only a representative sample of our seventeen applications. We refer the reader to [24] for an expanded version of this paper containing more comprehensive data.

---

[1] Note, however, that Dongarra and Dunigan have measured a peak bandwidth of only 8 MB/s on a KSR-1, which has 20-MHz processors connected by the same network as in the KSR-2 [10].

[2] All applications were measured while running default data sets that came with the benchmark suites, except that the number of iterations for QCD were reduced from 100 to 2 to shorten execution times in our experiments.

Application	Exec. time (secs)	Description
Barnes †	1159.14	Barnes-Hut N-body simulation.
Fft †	6.12	Fast Fourier transform.
Fmm †	602.98	Fast Multipole Method N-body simulation.
LocusRoute	78.56	VLSI standard cell router.
MP3D	25.02	Simulation of rarefied hypersonic flow.
Ocean †	1663.02	Model currents in ocean basin.
Pverify	52.30	Logical verification.
Raytrace †	271.90	Rendering of 3-dimensional scene.
Radix †	199.50	Integer radix sorting.
Water †	300.00	N-body molecular dynamics problem.

Table 1. *Hand-coded applications. († from SPLASH-2, remainder from SPLASH.)*

Application	KAP Exec. time (secs)	SUIF Exec. time (secs)	Description
ADM	364.12	–	Hydrodynamic simulation using mesoscale hydrodynamic model.
ARC2D	904.14	1699.48	Analysis of fluid flow problems using Euler equations.
DYFESM	175.94	327.48	Analysis of symmetric anisotropic structures.
FLO52	374.36	406.66	Analysis of transonic inciscid flow past an airfoil using unsteady Euler equations.
QCD	157.16	230.38	Simulation of gauge theory using a Monte Carlo-based algorithm.
TRACK	412.38	–	Tracking of moving targets based on sensor inputs.
USAero	3240.16	–	CFD computation.

Table 2. *Compiler-parallelized sequential applications. (All except USAero from PERFECT Club suite.)*

4 Speedup and Sources of Speedup Loss

4.1 Application Speedup

In this section, we examine the speedup characteristics of the jobs in our workload. We begin by giving the speedup functions for all jobs, both to better document the workload and to support previous work by others asserting that schedulers should take individual job speedups into consideration in making allocation decisions. We then identify and

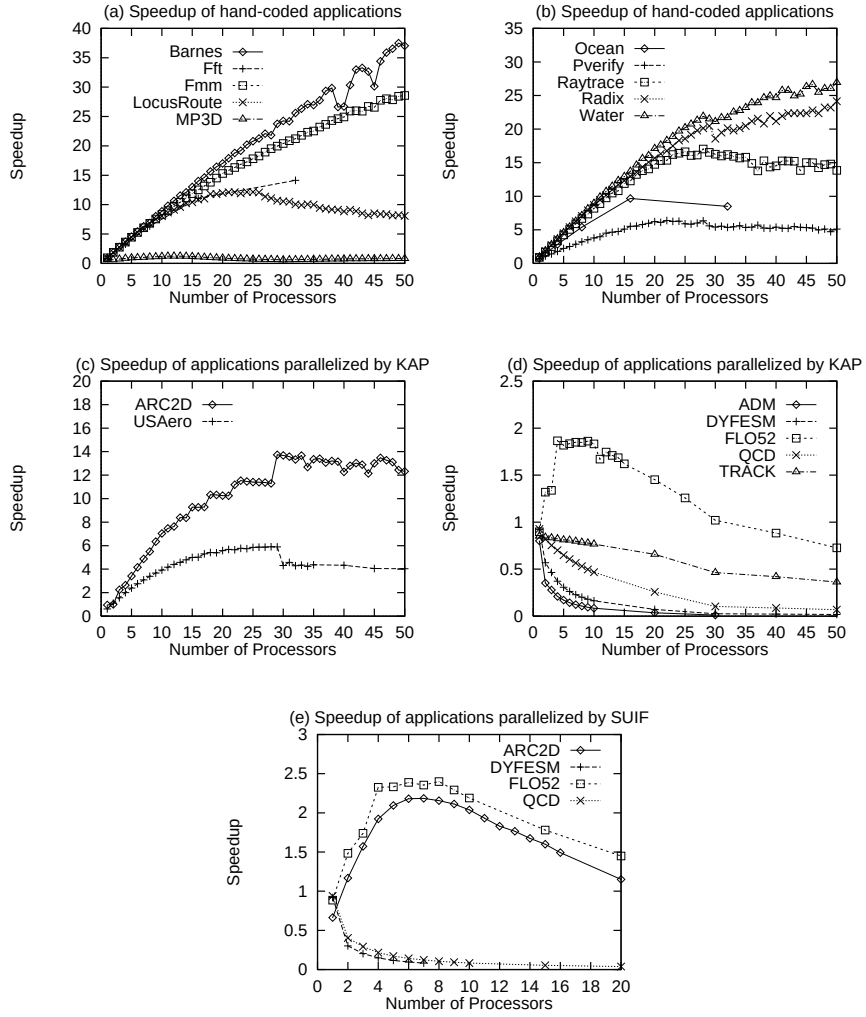

Fig. 1. *Speedup for (a), (b) hand-coded applications, (c), (d) sequential applications parallelized by KAP, and (e) sequential applications parallelized by SUIF. (Note: Different Y-axis scales are used for legibility.)*

quantify the sources of speedup loss as a prelude to an investigation of how speedup might be characterized through runtime measurement.

Figure 1 plots speedup against the number of processors. (The experiments for the applications parallelized by SUIF were terminated at 20 processors because the slowdown they experienced caused running times to be excessive. Also, not all applications could be run successfully with SUIF: either the compiler itself or the parallelized application failed during execution.) We observe that:

– Speedups vary greatly, even among applications in the same class (hand-coded or compiler-parallelized).
– Speedup is typically much worse for compiler-parallelized applications than for hand-coded applications.
– Most speedup curves are relatively smooth and roughly convex-shaped. This implies that speedup values for a relatively few allocations might allow reasonably accurate extrapolation to other allocations. (See [23] for an application of this idea to scheduling.)
– For most hand-coded applications, there is an allocation beyond which they slow down gradually. With the exception of ARC2D when parallelized by KAP, all compiler-parallelized jobs slow down significantly after achieving their peak speedups.
– Hand-coded applications seem to tolerate crossing the cluster boundary fairly well, whereas the two compiler-parallelized applications that were still speeding up at 30 processors (USAero and ARC2D), slow down when they are spread across clusters.

We now turn our attention to quantifying and characterizing factors contributing to speedup loss as applications are executed on larger processor allocations.

4.2 Loss of Speedup

We have two goals in this subsection. One immediate goal is to document the sources of speedup loss in our applications as part of our workload characterization. A second, longer term objective, is to work towards an accurate and efficient scheme for measuring speedup at runtime.

It is well-known that loss of speedup in shared-memory systems arise from the following factors [21, 28]:

1. **Idleness:** at times, parallel programs must idle allocated processors because of insufficient parallelism or load imbalance.
2. **Communication:** in shared-memory machines such as the KSR-2, communication takes place when the executing thread refers to data that either does not currently reside in its cache or is not in the appropriate state. In the case of the KSR-2, this can occur for both the processor cache and the attraction memory (which itself is a cache). KSR-2 processors stall while waiting for the data to be fetched from a remote node. Thus, on the KSR-2, communication overheads appear as processor stall.
3. **System overhead:** even sequential programs incur system overhead because of events such as page faults, clock interrupts, etc. Such overhead can be more significant for highly parallel programs, however, because these events typically occur on every processor (and so must occur more often for a program running on more processors). Furthermore, the asynchronous nature of these events can degrade the performance of tightly-coupled parallel programs.
4. **Parallelization overhead:** parallel programs typically must incur computational overheads that are not present in sequential programs, such as per-processor initialization, work partitioning, and locking and unlocking on entry and exit of a critical section.

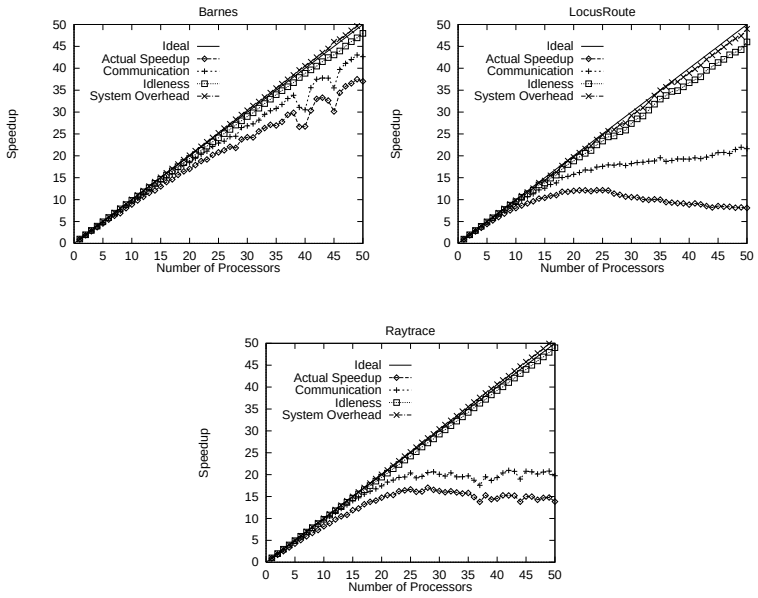

Fig. 2. *Loss of speedup for hand-coded applications. (The distance from each curve to the curve below it represents the speedup loss due to that factor.)*

Of the four sources, it is particularly difficult to measure parallelization overhead for hand-coded applications because initialization, work partitioning, and synchronization code are typically scattered throughout the application code. Thus, in what follows, we measure only idleness, communication, and system overhead, and infer parallelization cost from the remaining speedup loss.

Figures 2 and 3 plot speedup and speedup loss due to idleness, communication, and system overhead for a number of applications that achieve modest to good speedup. In each graph, the lowest curve represents actual measured speedup. Each curve above the speedup function represents what speedup would have been had a single source of speedup loss been eliminated. In order from bottom to top, we consider communication, idleness, system overhead, and parallelization overhead. (We also plot ideal (linear) speedup.) The graphs are cumulative; e.g., the curve for which idleness overhead has been set to 0 also has communication cost set to 0. Thus, the distance between each pair of curves in the figure indicates the magnitude of speedup loss due to the overhead associated with the higher curve. (Parallelization overhead is the difference between the ideal and system overhead curves.)

We observe that parallelization overhead is negligible, and that system overhead is typically very small compared to idleness and communication cost. On average, parallelization overhead accounts for less than 1% of application processor time while system overhead accounts for less than 3%. We note, though, that production workloads stressing the capacity of main memory could exhibit considerably more system overhead due to paging than we observe using these benchmark suites.

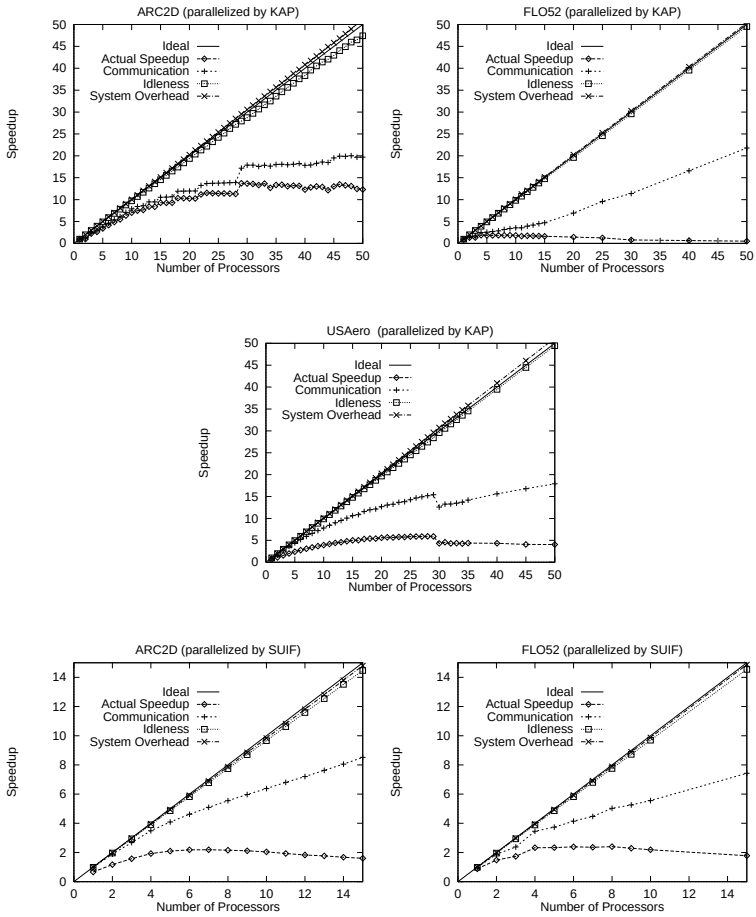

Fig. 3. *Loss of speedup for compiler-parallelized applications. (The distance from each curve to the curve below it represents the speedup loss due to that factor.)*

On the other hand, both hand-coded and compiler-parallelized applications can contain significant idleness, although compiler-parallelized applications tend to exhibit more. Among all the hand-coded applications we studied, idleness can be as high as 60% of processor time; even Barnes, the application showing the best overall speedup in our application suite, can contain as much as 20% idleness. KAP-parallelized applications can contain up to 83% idleness, while SUIF-parallelized applications can contain up to 50% idleness.

Similarly, both hand-coded and compiler-parallelized applications can contain significant communication overhead. Hand-coded applications seem to be divided into two classes, those that slow down and those that continue to speed up with increasing allocation size up to the maximum number of processors available. Interestingly, almost all those that slow down exhibit considerably worse communication losses than those that

do not (e.g., up to 94% of MP3D execution time can be attributed to communication overhead). Applications parallelized by either KAP or SUIF can contain up to 50% communication overhead.

5 Runtime Measurement of Speedup

In the previous section we observed that there is considerable variation in speedup behavior among jobs, encouraging the development of policies that use speedup information in making allocation decisions. Most speedup-sensitive policies that have been proposed to date have been static, and so require *a priori* specification of each job's speedup function. In this section we investigate a different approach that would be of use to dynamic policies, the acquisition of speedup characterizations at runtime through measurement. The attraction of using runtime measurements is both its convenience (since it relieves the user of the burden of providing this information) and its potentially greater accuracy (since applications whose speedup are sensitive to their input data or the relative locations of their allocated processors cannot be characterized *a priori*). Of course, runtime measurements can also be used to complement *a priori* information when such information is available.

We begin by looking at how accurate runtime speedup measurements can be made at reasonable overhead. We then examine the extent to which recent measurements of speedup predict future behavior.

5.1 Estimating Speedup Through Runtime Measurement

In the previous section we found that the majority of speedup loss is due to idleness and communication. In fact, Figures 2 and 3 show that there are very small differences between ideal speedup and the sum of actual speedup, idleness loss, and communication loss. As noted, however, system overhead can be more significant for programs with large memory requirements, where paging becomes a more significant source of overhead. These observations suggest that reasonably accurate measurements of speedup for scheduling purposes can be made at runtime by monitoring these three components of speedup loss.

We have implemented this approach to runtime measurement on our KSR-2. To measure communication cost and system overhead, we rely on hardware support. The per-node event monitors available on the KSR-2 (see Section 3.1) maintain three critical hardware counters: elapsed wall-clock time, elapsed user-mode execution time, and accumulated processor stall (communication). Dividing stall time by wall-clock time gives us the efficiency loss due to communication, from which we can infer the corresponding speedup loss. Dividing the difference between wall-clock time and user-mode time by wall-clock time gives us the efficiency loss due to system activities.

This method of computing communication cost and system overhead is quite efficient. If the scheduler is implemented in the operating system kernel, then accessing these counters is simply a matter of reading the appropriate hardware registers. If the scheduler is implemented as a user-level server, the hardware registers would need to be mapped to shared-memory or the server would have to make use of low-level messaging

services to read the remote registers. Note that in order for such a user-level server to read the remote registers, it would need to interrupt the running thread. If sample intervals are not too small, however, this is unlikely to be a significant source of overhead.

To measure idleness, we need to depend on the application itself. If applications are built using runtime systems such as Cthreads, then idleness measurements can be made without requiring explicit programmer effort by placing the measurement code in the thread package. Currently, we instrument the Cthreads synchronization code to keep running counts of processor idleness, and make this information available to the system via a piece of system-designated shared-memory. This approach is relatively overhead-free because idleness accounting is performed when the processor would otherwise not be doing any useful work. Of course, this approach assumes that all application synchronization takes place through calls to the CThreads libraries rather than through direct manipulation of shared variables. We did not, however, have to modify our applications to meet this assumption; none of our hand-coded programs violated this assumption, while all synchronization in compiler-parallelized applications by definition takes place in the thread package.

Note that while we have relied on the specific hardware counters on the KSR-2 processor, many modern processors include similar functionality. For example, both the DEC Alpha and the Intel Pentium processors contain counters for various sorts of cache misses, which could be translated into estimates of communication cost [31, 4].

5.2 Using Speedup Measurements to Predict Future Behavior

For runtime speedup measurements to be of practical use to schedulers, application speedups must be *predictable*. By predictable we mean that if an application's speedup is measured over some interval, the application will continue to execute at roughly that speedup for some time to come. In the remainder of this section we consider the question of application speedup predictability. To facilitate comparisons of prediction errors for different allocations of processors, we normalize our results. Specifically, we measure prediction errors in terms of efficiency, rather than speedup itself.

The simplest approach to predicting future speedup is to use quantum-based measurements of current speedup and to guess that the future will look like the past. Intuitively, we expect longer quantum lengths to result in more accurate predictions.

We evaluate this approach to speedup prediction through trace-based simulation. We first create traces for each of our applications. Each trace contains measured efficiencies for each $100ms$ of execution. Given these traces, we can compare the efficiency measured during each proposed *measurement quantum*, $Q > 100ms$, to the efficiency observed during the next quantum[3].

To evaluate the error of the efficiency predictions, we need to choose an appropriate measure. (For example, a natural choice might be mean absolute error.) Because we are interested in how useful these predictions will be to schedulers, choosing an appropriate

[3] Predicting that efficiency in the next quantum will be equal to efficiency in the just completed quantum is, of course, only one possible choice. We also investigated another natural choice, the use of exponentially decaying histories of all past observations. We found this technique to be generally less accurate, however.

measure is a difficult problem. On the one hand, the measure should reflect the average difference between the predicted and actual future efficiencies. On the other hand, the average difference alone is not sufficient information: it understates the error because occasional very incorrect predictions might induce a scheduler to make unfortunate allocation choices that can degrade performance much more than proportional to the error in the predictions (see, for example [22]). At the other extreme, looking at the maximum single-prediction error probably overestimates error, since errors of that magnitude may be exceedingly rare.

Because of the conflicting demands on the error measure to reflect both the common and the worst cases, we use a measure that can be parameterized to flow smoothly from one extreme to the other. In particular, let $M(Q)_i$ be the measured efficiency during the ith quantum of length Q, and let the complete execution consist of $N(Q)$ quanta. The measure of prediction error we use is

$$Error \;=\; \sqrt[C]{\frac{1}{N(Q)-1}\sum_{i=1}^{N(Q)-1}\left(|M(Q)_{i+1}-M(Q)_i|\right)^C} \qquad (1)$$

For $C = 1$, this is the mean absolute error; as $C \rightarrow \infty$, the measure increasingly reflects the maximum absolute error.

Figures 4 and 5 graph our error measure for a selected subset of our applications. In each graph, the X-axis represents the measurement quantum length, Q, in ms, and the Y-axis the error measure. The distinct curves on each graph correspond to different values of C, the parameter of our measure.

We make three observations based on this data. First, for some applications, even quite long measurement intervals are not sufficient to obtain good accuracy, while for others much shorter intervals suffice. This makes choosing a system measurement interval difficult: long intervals are needed for some jobs, but shorter intervals are more advantageous to the scheduler (since they allow more frequent opportunities to correct inappropriate allocations).

Second, for all applications, the accuracy at a fixed measurement interval improves as the application is allocated more processors. This is simply a reflection of a changing time scale: the job is able to execute a larger fragment of its code in a fixed time period such that the time period becomes more representative of overall behavior.

Finally, many of the applications exhibit mild periodic behavior in accuracy as a function of quantum length.

All of these observations have a common explanation: there is little reason to expect the next measurement interval to look like the previous one unless the application is executing substantially similar code in both. Stated differently, we expect efficiency measurements to be most accurate in predicting future behavior when the measurement interval corresponds to the execution of some section of code that will be repeated.

In related work [22, 23], we have made use of this observation, exploiting a particularly simple (but also quite common) program structure: an outer sequential loop that drives the execution. In those works we show that schedulers that use measurements taken over intervals corresponding to executions of the outer loop can significantly improve application as well as system performance.

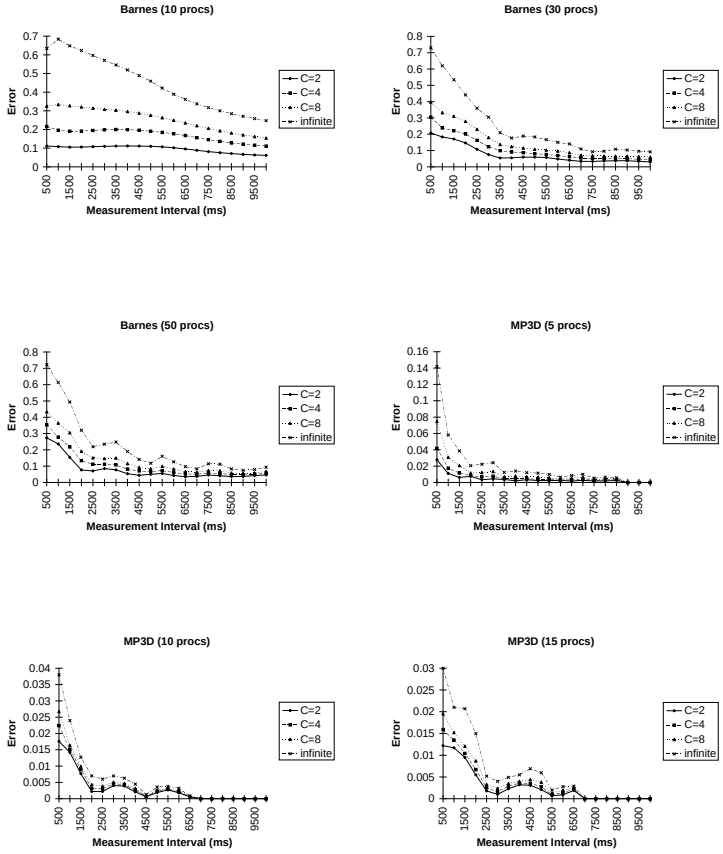

Fig. 4. *Prediction Error Versus Measurement Quantum Length (Q). (Note: Different Y-axis scales are used for legibility.)*

Of course, having to rely on a particular program structure (as well as the cooperation of the application to indicate when it has reached the beginning of an iteration) is not ideal. It remains to be seen whether it is possible to design schedulers that can accurately predict speedup without having to depend on application cooperation. Our data suggests that such a scheduler would have to dynamically "learn" the appropriate quantum for individual applications.

6 Processor Idle Periods

Multiprocessor scheduling disciplines can be broadly characterized as being static (the allocation made to a job is kept fixed for its entire execution), quasi-dynamic (realloca-

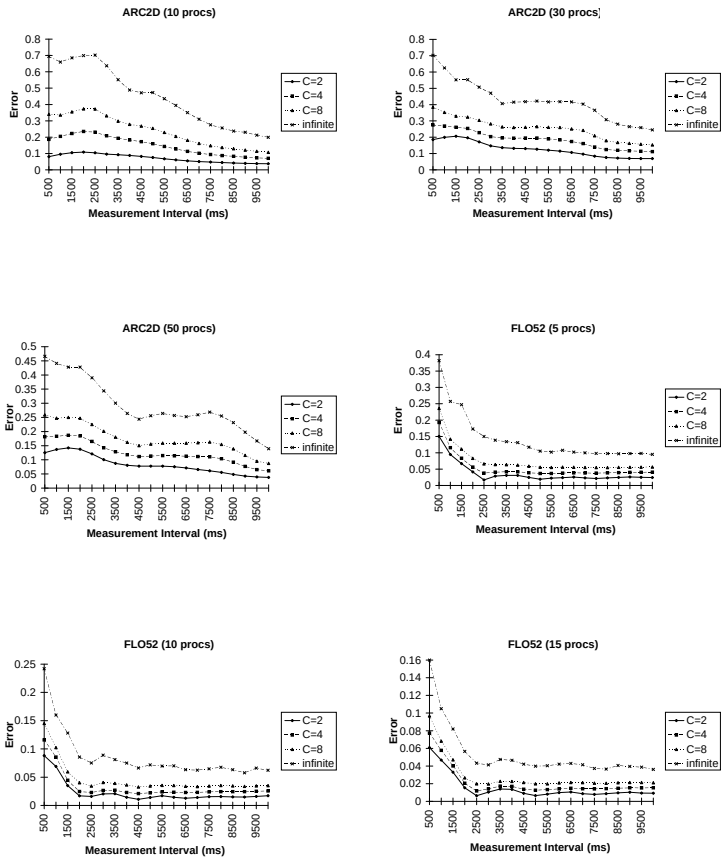

Fig. 5. *Prediction Error Versus Measurement Quantum Length (Q). (Note: Different Y-axis scales are used for legibility.)*

tions are performed at job arrival and departure moments, but not otherwise), or dynamic (reallocations may be performed at any time). In this section we focus on application characteristics that reflect on the opportunity to improve performance through dynamic reallocations.

The primary potential advantage of dynamic policies over quasi-dynamic ones is their ability to exploit the processor idleness that occurs during execution due to such things as sequential portions of execution, load imbalance, and contention for critical sections. The speedup results given in Section 4 show that there is considerable idleness in typical applications. To take advantage of this idleness, however, the duration of individual idle periods must exceed the cost associated with reallocating the processor. In the next subsection, we examine the reallocation cost on our machine. In succeed-

From	Fill Time
local attraction memory	1.28ms
within cluster	8.96ms
outside cluster	30.72ms

Table 3. *Time required to completely refill the KSR-2's 256KB processor cache.*

ing subsections we present measurements providing indications of whether processor idleness can be exploited through dynamic reallocation.

6.1 Processor Reallocation Cost

There are two components to processor reallocation cost: *system path length* and *cache penalty*. System path length is the time required to execute operating system code for reassigning a processor from one job to another, i.e., to perform a context switch. Measurements on our KSR-2 shows path length context switch costs in the range of 3 to 5ms. However, the KSR-2 processor reallocation mechanism was designed for ease of implementation, and uses a simple but very inefficient approach. In contrast, measurements of a Sequent Symmetry, an older shared-memory multiprocessor with much slower processors, indicate path length costs for context switching of about 750μs [20]. Based on this somewhat conflicting information, it appears that context switch path length costs below 1ms are easily possible on modern multiprocessors. However, it is unlikely that designers of production systems will invest the effort to optimize context switching until it becomes clear that there is a tangible payoff. For this reason, a conservative estimate of 1-2ms context switch times may be a reasonable reflection of typical systems.

The cache penalty component of reallocation cost reflects the fact that dynamic movement of processors can adversely affect program cache behavior, and therefore performance. The importance of cache performance to modern processor speed has motivated the recent work on cache-affinity scheduling [33, 32, 14, 35].

To evaluate the cache related cost of dynamic reallocation, we look at the worst-case times on the KSR-2. Recall that each processor in the KSR-2 has two caches, a 256 KByte processor cache and a 32 MB attraction memory. In what follows we focus on the processor cache, as we believe it will be the major source of cache interference over the reallocation intervals we are considering. A miss in the processor cache can be filled by three levels in the memory hierarchy: the local attraction memory, the attraction memory of another processor on the same cluster, and the attraction memory of a processor on another cluster. Table 3 gives the times required to completely fill the 256 KByte processor cache from these three levels of machine memory.

6.2 Idle Period Length Distribution

We consider first the distribution of the length of processor idle periods. Figure 6 gives results for five applications that achieves moderate to good speedup. For each number

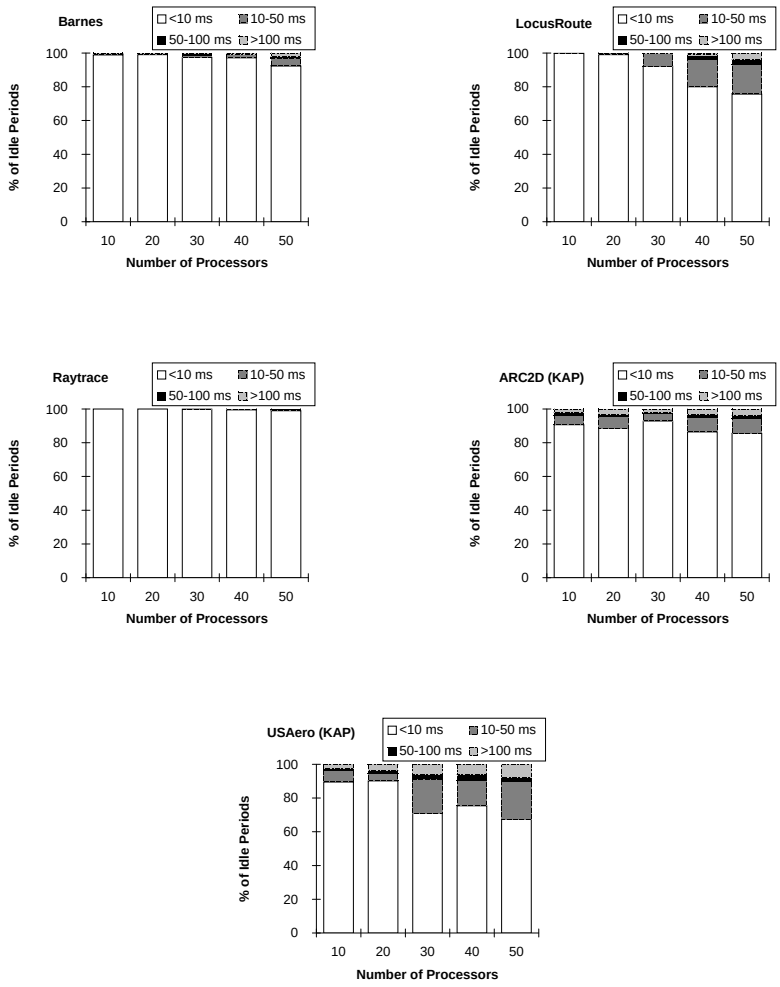

Fig. 6. *Distribution of idle period lengths. (Each component of the bars is the fraction of the total number of idle periods whose lengths fall in the intervals given in the legend.)*

of processors we show the percentage of all idle periods with durations that fall into four intervals: (0-10ms), [10-50ms), [50-100ms), and $>$100ms.

These graphs show that an overwhelming number of idle periods are short – less than 10ms. Furthermore, more detailed examination of the data shows that the average length of these short idle periods is typically well below 1ms. This suggests that aggressive dynamic reallocation (e.g., reallocating at the beginning of every idle period) may be ineffective or even detrimental to system performance for both application classes,

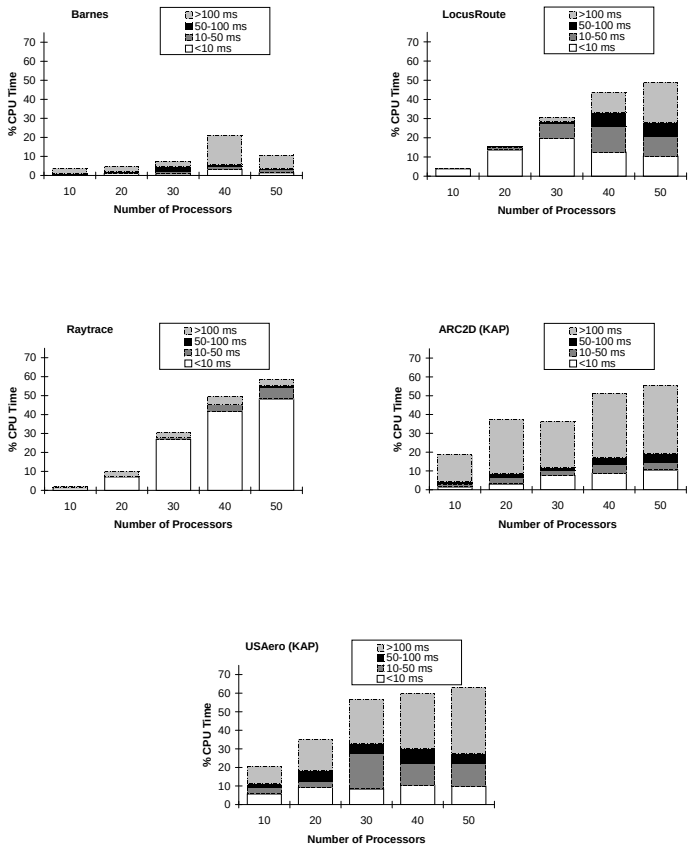

Fig. 7. *Distribution of idle time. (Each component of the bars is the fraction of processor time represented by idle periods whose lengths fall in the intervals given in the legend.)*

since estimated reallocation cost (Section 6.1) is longer than the length of the typical idle period for either class. This set of results for compiler-parallelized applications were particularly surprising, suggesting that sequential portions typically run for only short periods of time.

6.3 Idle Period Time Distribution

One approach to dealing with short idle periods is to filter them by waiting a short time before context switching. Ousterhout [25], Lo and Gligor [18], and Karlin et al. [15] take this approach in the context of implementing locks for mutual exclusion, where such filtering is called “two-phase blocking” or “spin-then-block.” McCann et. al. [20] have proposed a delayed reallocation scheme as part of a dynamic scheduling policy.

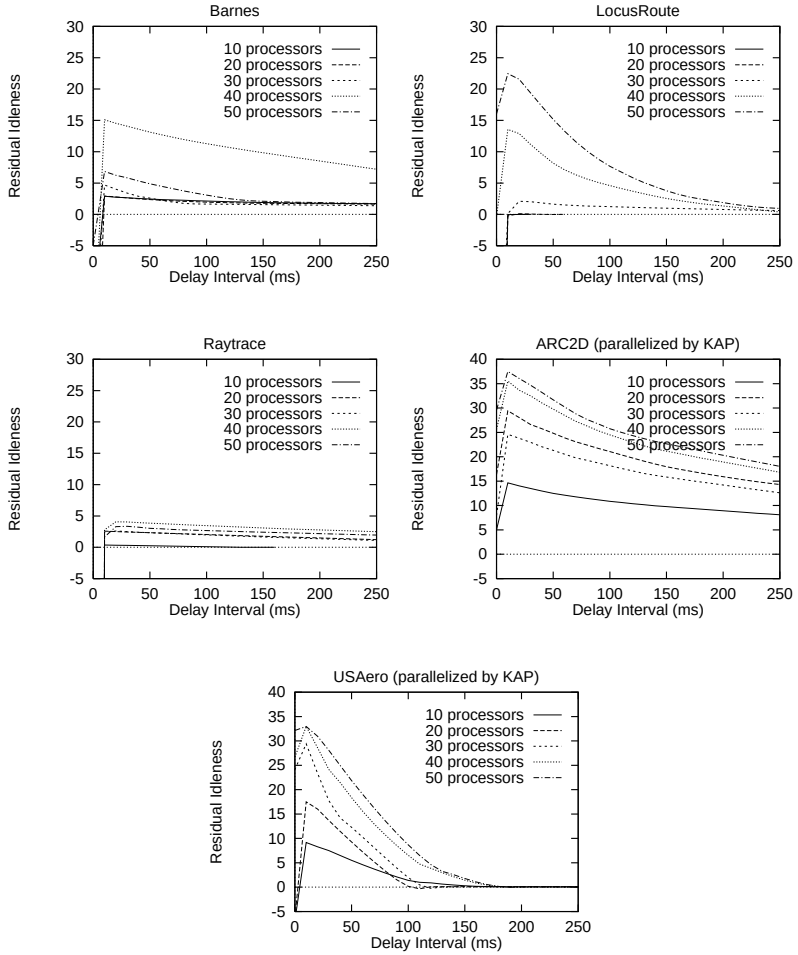

Fig. 8. *Residual idleness as a percentage of processor time assuming a 10ms total reallocation cost.*

A natural question to ask is how much idleness exists whose duration exceeds the context switch delay time. We address this question in Figure 7. Each bar shows the percent of total processor time represented by idle periods with durations in (0-10ms), [10-50ms), [50-100ms), and $>$100ms. This figure shows that long idle periods, although few in numbers, can account for a large fraction of total processor time, providing evidence that delayed reallocation may be profitable, at least for many applications.

6.4 Residual Idleness After Filtering

A delay-based reallocation scheme can improve performance only if application *residual idleness* – idleness remaining after the delay interval has elapsed – exceeds the cost of

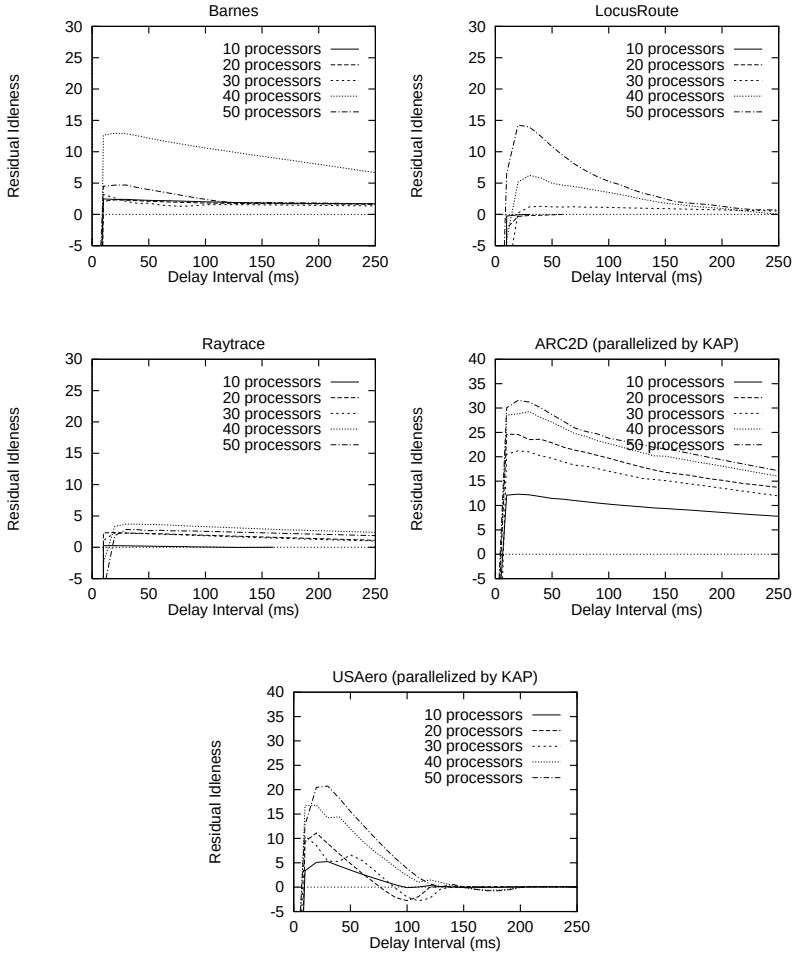

Fig. 9. *Residual idleness as a percentage of processor time assuming a $30ms$ total reallocation cost.*

reallocating the processor. Thus, we now consider residual idleness.

In what follows, we compute residual idleness by subtracting the sum of the delay time and the reallocation time from the total duration of each idle period greater than the delay time. The reallocation time includes two reallocation path length costs and two cache fill times. Because the latter depends strongly on both the workload (because of the footprints of the original and replacing applications) and on the system (because data placement decisions affect cache refill time), we use a number of different assumptions about the total cost to reflect different possible scenarios. We use $10ms$ as a conservative estimate for the case that either footprints are small or else the cache can be refilled from the local attraction memory. We use $70ms$ as an estimate for the case of very large footprints that must be filled from memories not on the local cluster. Finally, we

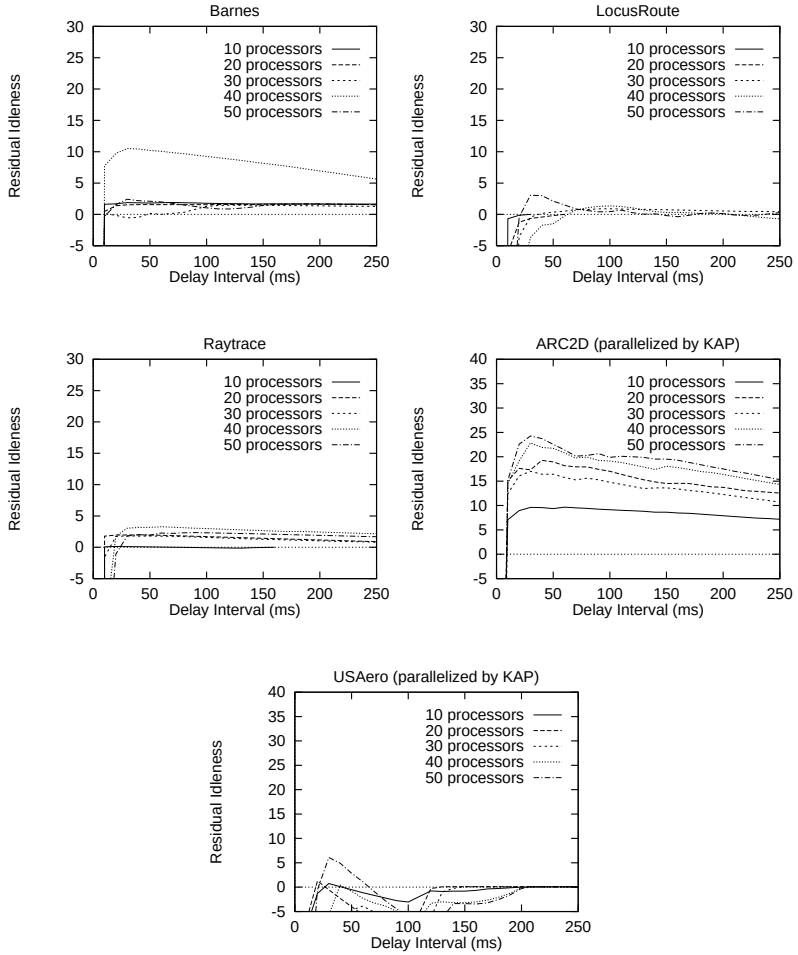

Fig. 10. *Residual idleness as a percentage of processor time assuming a 70ms total reallocation cost.*

use 30ms as a compromise between these extremes, representing perhaps a more likely cost.

Figures 8 and 9 plot residual idleness as a function of the delay before reallocation, assuming a total reallocation cost of 10ms and 30ms respectively. The Y-axis value of each point on each curve shows residual idleness as a fraction of total job processor time. The initial rise in the curves represents the benefit of a short delay – uselessly short idle periods are filtered out. The gradual decrease in the curves for longer delays represents the lost opportunity to use the processor as the delay to reallocate increases.

These figures show that delays from 10ms to about 20ms can result in significant recovery of processing power in many applications, while for the rest there is little or no loss. We also note that the appropriate delay time depends only weakly on the reallocation

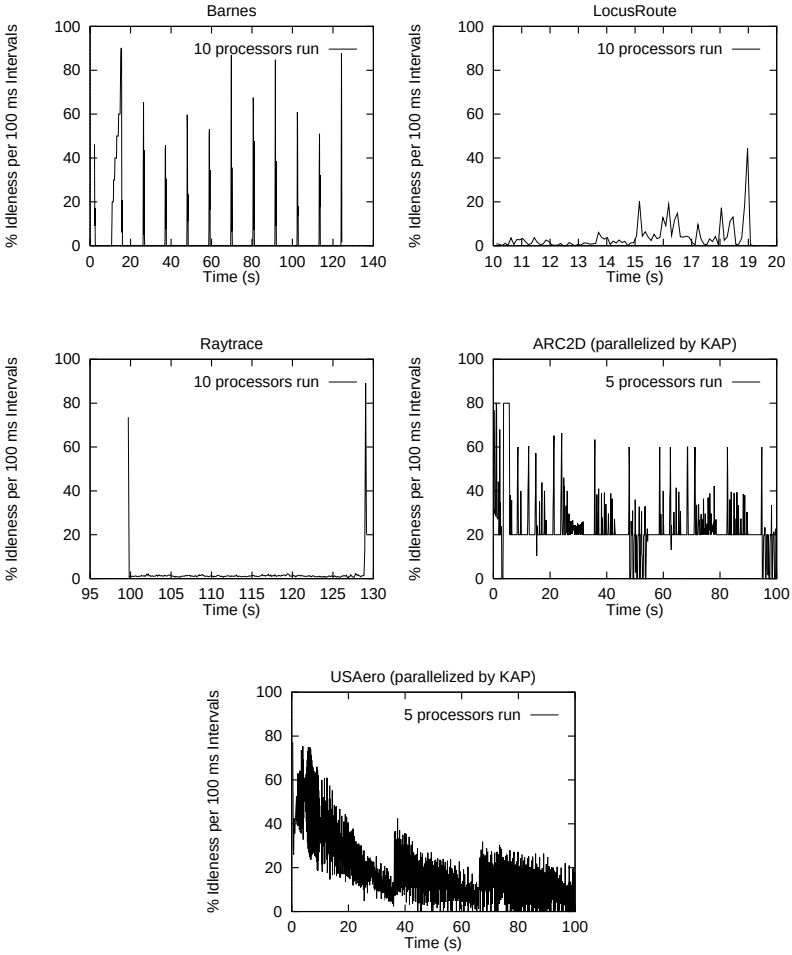

Fig. 11. *Idleness (measured over 100ms intervals) versus time.*

cost, indicating that it is more strongly tied to characteristics of the applications (i.e., the distribution of idle period lengths) than to reallocation times. At these reallocation costs, the distinctions in distributions among the applications are remarkably small; all give acceptable results for similar delay times.

Figure 10 plots normalized residual idleness assuming a reallocation cost of 70ms. At this cost, dynamic reallocation becomes less attractive. The previously productive delay range (10-20ms) can seriously degrade performance for some applications, although dynamic reallocation continues to recover significant processing power for others.

6.5 Using Idle Periods to Run Interactive Jobs

The previous subsection has shown that idleness-based dynamic reallocation can improve system performance, especially if incurred cache penalties are small and receiving threads can effectively make use of short processing periods. For this reason, we speculate that it may be more profitable to reallocate an idle processor to an interactive job rather than a parallel one. While current proposals for space-sharing systems that address support of interactive work typically partition available processors into two pools, one to run interactive jobs and one to run parallel jobs [34, 2], taking advantage of processors idled by parallel jobs could reduce the size of the partition dedicated to interactive work. Of interest to such schemes is whether idle periods occur often enough to support the running of interactive jobs without degrading the response time observed by interactive users.

Figure 11 plots idleness timelines, that is, "instantaneous idleness" against application execution time. We show timelines for hand-coded applications running on 10 processors and compiler-parallelized applications running on 5 processors. Timelines for different number of processors are similar to those shown. We observe that while hand-coded applications can contain long compute periods between idle periods, compiler-parallelized applications display almost continuous idleness. Thus, while we have not yet pursued this proposal, it seems plausible that at least some of the support for interactive computing could be provided by making use of temporarily idle processors allocated to parallel jobs.

7 Conclusions

In this paper, we have presented measurements of the behavior of two distinct implementation classes of scientific applications on a shared-memory multiprocessor system. Based on our measurements, we make the following observations:

- Significant differences exist in the speedup behavior of applications, supporting the importance of work on scheduling policies that use speedup information to guide scheduling decisions.
- For systems where *a priori* speedup information is not available, in many cases, it is possible to characterize general application behavior using runtime measurements of idleness, communication, and system overhead.
- On shared-memory systems, in order to reliably predict application speedups at runtime, the scheduler must rely on information provided by the runtime system.
- Although most idle periods are too short to merit reallocation of the processor, for some applications, long idle periods represent a significant fraction of execution time. We have shown that, for these applications, imposing a short delay before reallocating a processor when it goes idle is effective in filtering out the short idle periods and recovering much of this idleness.

Acknowledgments

Mary Vernon provided insightful comments that helped with both the content and presentation of this work. We thank Analytical Methods, Inc. for providing the USAero CFD application.

References

1. A. Agarwal and A. Gupta. Memory-Reference Characteristics of Multiprocessor Applications under MACH. In *Proceedings of the ACM SIGMETRICS Conference*, pages 215–225, May 1988.
2. I. Ashok and J. Zahorjan. Scheduling a Mixed Interactive and Batch Workload on a Parallel, Shared Memory Supercomputer. In *Supercomputing '92*, pages 616–625, Nov. 1992.
3. M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Scharzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, 1989.
4. J. Chen, Y. Endo, K. Chan, D. Mazieres, A. Dias, M. Seltzer, and M. Smith. The Measured Performance of Personal Computer Operating Systems. In *Proceedings of the 15th ACM Symposium on Operating system Principles*, pages 299–313, Dec. 1995.
5. S.-H. Chiang, R. K. Mansharamani, and M. K. Vernon. Use of Application Characteristics and Limited Preemption for Run-To-Completion Parallel Processor Scheduling Policies. In *Proceedings of the ACM SIGMETRICS Conference*, pages 33–44, May 1994.
6. E. C. Cooper and R. P. Draves. C Threads. Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie-Mellon University, June 1988.
7. G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer Performance Evaluation and the Perfect Benchmarks. In *Proceedings of the 1990 International Conference on Supercomputing, ACM SIGARCH Computer Architecture News*, pages 254–266, Sept. 1990.
8. R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural Requirements of Parallel Scientific Applications with Explicit Communication. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 2–13, May 1993.
9. F. Darema-Rogers, G. Pfister, and K. So. Memory Access Patterns of Parallel Scientific Programs. In *Proceedings of the ACM SIGMETRICS Conference*, pages 46–58, May 1987.
10. J. J. Dongarra and T. Dunigan. Message-Passing Performance of Various Computers. Technical Report CS-95-299, University of Tennessee, July 1995.
11. R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the Parallelization of Four Perfect-Benchmark Programs. Technical Report 1193, Center for Supercomputing Research and Development, Aug. 1991.
12. D. G. Feitelson and B. Nitzberg. Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860. In *Proceedings of the IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 337–360, Apr. 1995.
13. K. Guha. Using Parallel Program Characteristics in Dynamic Processor Allocation Policies. Technical Report CS-95-03, Department of Computer Science, York University, May 1995.
14. A. Gupta, A. Tucker, and S. Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *Proceedings of the ACM SIGMETRICS Conference*, pages 120–133, May 1991.

15. A. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 41–55, Oct. 1991.
16. Kendall Square Research Inc., 170 Tracer Lane, Waltham, MA 02154. *KSR Fortran Programming*, 1993.
17. S. T. Leutenegger and M. K. Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Policies. In *Proceedings of the ACM SIGMETRICS Conference*, pages 226–236, May 1990.
18. S.-P. Lo and V. Gligor. A Comparative Analysis of Multiprocessor Scheduling Algorithms. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 356–63, Sept. 1987.
19. S. Majumdar, D. L. Eager, and R. B. Bunt. Scheduling in Multiprogrammed Parallel Systems. In *Proceedings of the ACM SIGMETRICS Conference*, pages 104–113, May 1988.
20. C. McCann, R. Vaswani, and J. Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.
21. A. J. Musciano and T. L. Sterling. Efficient Dynamic Scheduling of Medium-Grained Tasks for General Purpose Parallel Processing. In *Proceedings of the International Conference on Parallel Processing*, pages 166–175, Aug. 1988.
22. T. D. Nguyen, R. Vaswani, and J. Zahorjan. Maximizing Speedup Through Self-Tuning of Processor Allocation. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 463–468, Apr. 1996.
23. T. D. Nguyen, R. Vaswani, and J. Zahorjan. Using Runtime Measured Workload Characteristics in Parallel Processor Scheduling. In *Proceedings of the IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*, Apr. 1996.
24. T. D. Nguyen, R. Vaswani, and J. Zahorjan. Parallel Application Characterization for Multiprocessor Scheduling Policy Design. Technical report, Department of Computer Science and Engineering, University of Washington, In preparation.
25. J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of 3rd International Conference on Distributed Computing Systems*, pages 22–30, Oct. 1982.
26. P. Petersen and D. Padua. Machine-Independent Evaluation of Parallelizing Compilers. Technical Report 1173, Center for Supercomputing Research and Development, 1992.
27. E. Rothberg, J. P. Singh, and A. Gupta. Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 14–25, May 1993.
28. K. C. Sevcik. Characterizations of Parallelism in Applications and their Use in Scheduling. In *Proceedings of the ACM SIGMETRICS Conference*, pages 171–180, May 1989.
29. K. C. Sevcik. Application Scheduling and Processor Allocation in Multiprogrammed Parallel Processing Systems. *Performance Evaluation*, 19(2/3):107–140, Mar. 1994.
30. J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, 1992.
31. R. L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.
32. M. Squillante and E. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.
33. D. Thiebaut and H. S. Stone. Footprints in the Cache. *ACM Transactions on Computer Systems*, 5(4):305–329, Nov. 1987.
34. A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 159–166, Dec. 1989.

35. R. Vaswani and J. Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 26–40, Dec. 1991.
36. R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Comilers. Technical report, Computer Systems Laboratory, Stanford Univeristy.
37. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, , and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.