Adaptive Parallelism in Compiler-Parallelized Code

Mary W. Hall*
USC Information Sciences Inst.
4676 Admiralty Way
Marina del Rey, CA 90292
mhall@isi.edu

Margaret Martonosi
Dept. of Electrical Eng.
Princeton University
Princeton, NJ 08544-5263
mrm@ee.princeton.edu

Abstract

As moderate-scale multiprocessors become widely used, we foresee an increased demand for effective compiler parallelization and efficient management of parallelism. While parallelizing compilers are achieving success at identifying parallelism, they are less adept at pre-determining the degree of parallelism in different program phases. Thus, a compiler-parallelized application may execute on more processors than it can effectively use, a waste of computational resources that becomes more acute as number of processors increases, particularly for systems used as multiprogrammed compute servers.

This paper examines the dynamic parallelism behavior of multiprogrammed workloads using programs from the Specfp95 and NAS benchmark suites, automatically parallelized by the Stanford SUIF compiler. Our results demonstrate that even the programs with good overall speedups display wide variability in the number of processors each phase (or loop) can exploit. We propose and evaluate a run-time system mechanism that dynamically adjusts the number of processors used by a compiler-parallelized application, responding to observed performance *during* the program's execution. Executing programs can thus adapt processor usage both to poor parallelism within certain parts of their code, and also to heavy multiprogramming loads during the execution. This mechanism *improves workload performance up to 33%* over consecutive standalone runs of each program.

*Corresponding author, Phone: (310) 822-1511, x254, FAX: (310) 823-6714

1 Introduction

Moderate-scale parallel computers are more and more commonly being used as general-purpose compute servers, increasing the value of parallel programming tools such as parallelizing compilers. Years of work on parallelizing compilers for C and FORTRAN has brought them to a stage where they successfully identify parallelizable loops in many codes. Their success in identifying parallelism opportunities has led to fairly impressive speedups on four and eight processor runs [3, 1]. Compiler-parallelized applications are not as consistently successful, however, at yielding near-optimal speedups on larger parallel machines; as the number of processors increases, fewer and fewer applications have sufficient parallelism granularity to make effective use of all the processors.

Thus, if compiler-parallelized applications are to be run on larger-scale parallel machines, the issue of effectively mapping the parallel computations to the processors becomes increasingly important. Today's parallelizing compilers typically map parallel loops to processors statically. Most commonly, they simply assign all the machine's processors to the computation. Because speedups will typically level off (or even drop!) as processors are added, additional processors often do little for program performance and might be better used elsewhere. This paper demonstrates that a significant waste of processor resources can be avoided by tailoring the mapping of a parallel computation to the number of processors it can use effectively. Furthermore, since parallelism changes in different program phases, this mapping may best be done intermittently at run time, rather than once at compile time.

Determining the appropriate degree of parallelism in program phases is important for both program and system performance. At the system level, we expect that many moderate-scale machines will be used as multiprogrammed compute servers, running a mix of sequential and parallel jobs. If each computation uses only the number of processors it needs to approach its maximum speedup, the overall system can maintain high throughput by not wasting computational resources while still observing the low latency benefits of parallelizing individual applications. For individual programs, mappings involving fewer processors will sometimes reduce false sharing and improve performance. In the future, accurate mappings will become even more important as compiler-parallelized applications become increasingly complex, and incorporate more general multi-threading and combined task/data-parallelism constructs.

An appropriate processor mapping requires accurate knowledge about program parallelism. Estimating a program's per-computation parallelism at compile-time is often difficult, due to variable loop bounds, unpredictable memory behavior, and other effects [13]. For this reason, this paper argues for the need for *dynamic* mapping of computations to processors, which not only offers more accuracy than static mappings but can also adapt to multiprogrammed workloads. Through a dynamic threads package (extending the SUIF run-time system [16]), our approach adaptively manages a pool of threads for multiple programs; it gradually deallocates threads from programs making ineffective use of parallelism, and allocates them to programs making effective use of parallelism based on run-time performance measurements.

While there have been other dynamic approaches to parallelism management in multiprogrammed environments, this work is distinguished by two features. First, it is fully automatic and completely transparent to the user. Second, it uses more sophisticated information— dynamic measurements of how effectively an individual computation is using its assigned processors—to guide allocation. Overall this work contributes (i) practical observations on loop-level behavior of compiler-parallelized code, (ii) a careful exploration of policy issues in run-time parallelism management, and (iii) data evaluating a prototype implementation such a system in a moderate-scale shared-memory multiprocessor.

The remainder of this paper is structured as follows. In Section 2, we describe the SUIF parallelizing compiler and run-time system upon which we base our work. Section 3 presents overall program characteristics and per-loop data for five common benchmarks, which illustrates the variability in parallelism over the course of program execution. Based on these observations, Section 4 discusses our mechanism for adaptive thread management and fine-grained time-sharing. Section 5 presents experimental results for a proof-of-concept implementation. Section 6 presents related work and Section 7 concludes with a discussion of requirements for this approach and future extensions.

2 SUIF Compiler Parallelization and Run-Time System

The programs in our experiment were parallelized using the Stanford SUIF compiler [16]. SUIF takes as input sequential Fortran or C programs, producing as output parallel SPMD C programs, and has demonstrated significant overall speedup improvements on realistic benchmark programs [3].

The SUIF compiler determines the outermost loop in a loop nest for which it is safe to perform parallelization. The iterations of an outermost parallel loop are divided at compile time so that each processor performs a roughly equal number of consecutive iterations. SUIF programs rely on a run-time system built from ANL macros for thread creation, barriers, and locks. The run-time system creates P threads, where one is a master thread and the others are worker threads that spin at a barrier waiting for work from the master.

The standard interaction between the SUIF compiler and the run-time system is as follows. SUIF generates sequential and parallel versions of all outermost parallelizable loops. The compiler and run-time system then collaborate to "turn off" parallel execution of certain loops that they determine do not have sufficient granularity for profitable parallelization, executing the sequential version of the loops instead. This decision is based on static information regarding the amount of work in a single iteration of the loop and dynamic information on number of loop iterations. Thus, a loop executes either sequentially or on all the available processors.

SUIF's mostly static, all-or-nothing strategy has several limitations. First, static models cannot easily capture effects of memory behavior, which can be very complex and can significantly affect performance. Second, parallel loops may behave differently throughout a program's execution — such as when number of iterations of a loop varies across invocations — so a solution appropriate to one invocation may not be adequate for another. Third, the all-or-nothing approach does not scale. As larger numbers of processors are applied to the application, fewer loops will have sufficient granularity to take advantage of *all* the processors, and thus, less and less of the application will execute profitably in parallel.

Limitations in SUIF's standard static approach motivate our adaptive run-time system, which uses dynamic information to guide mapping parallel loops to processors. By adjusting the number of processors used for each loop according to its requirements, we can make more efficient use of parallelism on larger numbers of processors.

3 Program Characteristics

Program	Suite	Description	Input Data Set	Iters.	14-proc. Speedup
cgm	NAS	sparse conjugate gradient	14000 array elements	14000	5.7
hydro2d	Specfp95	Navier-Stokes	402x160 grid	20	7.8
mgrid	Specfp95	multigrid solver	64^3 grid	25	10.9
su2cor	Specfp95	quantum physics	12^4 grid	40	2.9
swim	Specfp95	shallow water model	512^2 grid	128	13.0

Table I: Characteristics of scientific applications in study.

Our results focus on workloads comprised of combinations of a set of five benchmarks. Four of the five programs—hydro2d, mgrid, su2cor and swim—are taken from the suite of Specfp95 floating-point benchmarks [12]. The remaining program, cgm, is from the NAS sample benchmarks [7]. These programs were chosen for two main reasons. First, they are real computational benchmarks, not toy programs or toy data set sizes. Second, taken together they represent a range of computation granularities with varying parallelism behavior. Table I briefly describes features of the applications.

We used standard data set sizes for all of the programs, using the "test" inputs for the Specfp95 programs and the large input for cgm. Other than swim, the programs had sequential execution times ranging from 3.5 to 8 minutes. Because swim's "test" input executed in under a minute, we modified the iteration count for swim to bring its 1-processor execution time up to 3.5 minutes, similar to other applications. This one small change was needed so that our experiments measured the behavior of executing programs with comparable execution times together.

3.1 Overall Application Parallelism

The rightmost column of Table I summarizes application speedups on an SGI PowerChallenge multiprocessor running Irix 6.2. The machine had a total of 18 R8000 processors. Since the machine was configured with a mix of 14 75Mhz and 4 90MHz processors, our experiments only show data up to 14 processors, and we assigned processes to processors such that the four faster processors were never used for the measured applications.

The SUIF compiler does a fairly good job of finding and exploiting parallelism for these applications, but these programs span a wide range of parallelism granularities and, as a consequence, overall speedups. We observe speedups ranging from a near-linear 13.0 for swim down to a modest 2.9 for su2cor.

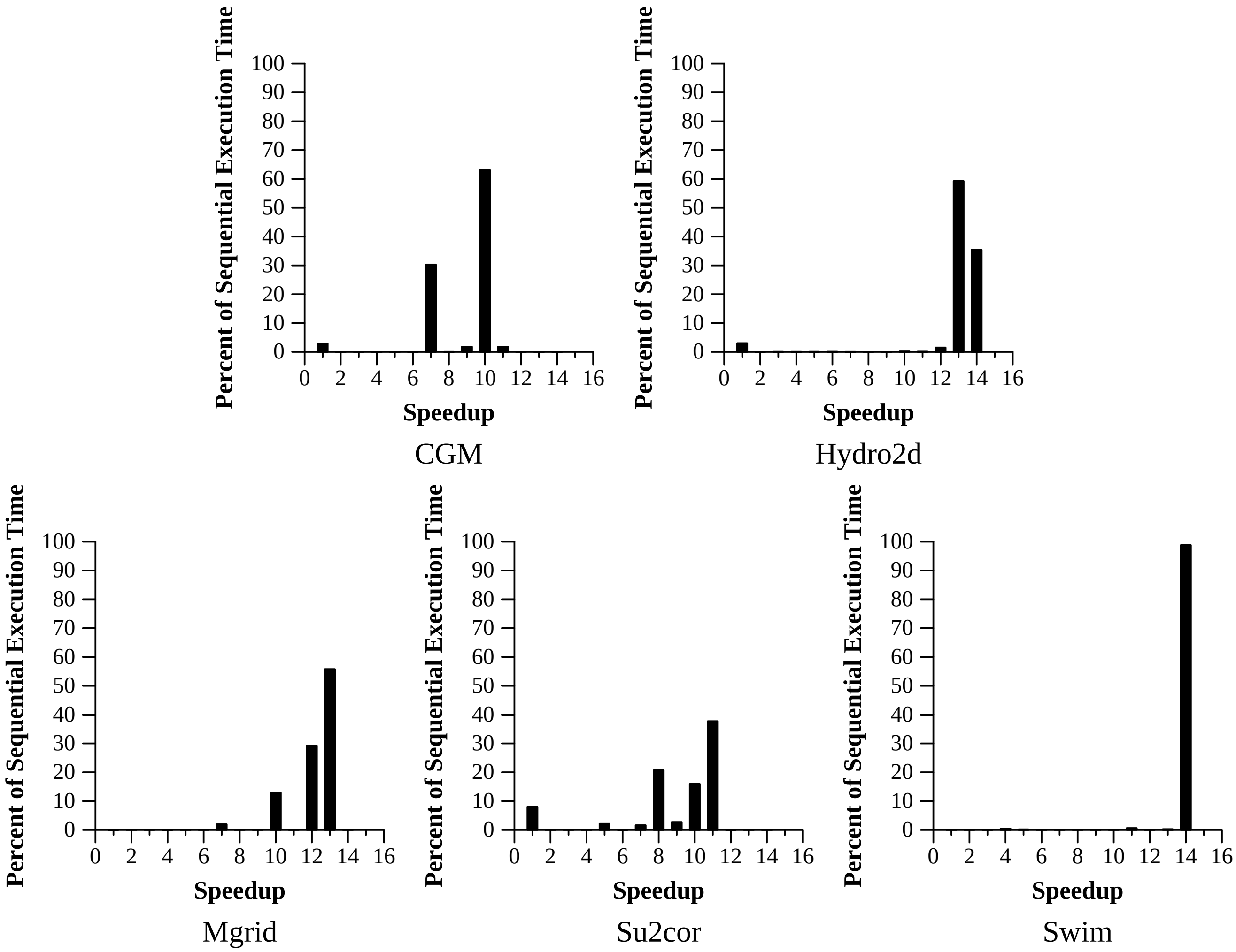

Figure 1: Parallelism distribution for each of the applications.

3.2 Loop-level Performance

We shed light on the speedups in Table I by considering application performance on a per-loop basis. For each application, Figure 1 shows a histogram weighting the time the application spends at different speedups. To gather these numbers, we first ran the program sequentially, and measured both its overall execution time and also the time spent in each loop. The entire program was then re-run with a full 14 processors and once again we measured the time spent in each loop. From these two runs, we can calculate individual speedups for each loop in the program, and histogram them to show how often different speedup values occur. The x-axis of the graph is divided into histogram bins corresponding to speedups of 1.0 or less, 1.0+ to 2.0, and so on. Each bar corresponds to the total fraction of the program's *sequential execution* time spent on code with a particular speedup.

Overall, the figure shows that within each application, there is noticeable variability in the speedups at which they spend significant execution time. Swim, the application with the best performance, has the least variability. Both hydro2d and mgrid, however, have loops with speedups near ideal but also loops with much lower speedups. In the case of cgm, the effect is more dramatic. One loop in the program is responsible for 28.6% of the sequential execution time, and yet has a speedup of only 6.9 with 14 processors because of a small granularity of parallelism. The program with the most widely differing behavior is su2cor, with a large sequential component and parallel loops with speedups ranging from 8 to 11.

The significant speedup variability noted in all the programs except swim indicates that compiler-parallelized programs are frequently not making efficient use of the processors allocated to them. We expect the majority of moderate-scale multiprocessors to be used as multiprogrammed compute servers; thus, if one program is sometimes using processors inefficiently, it is natural to consider shifting processors *away* from this program, and *towards* programs that might use them more effectively. Our work explores mechanisms for dynamically managing threads to accomplish this.

4 Adaptive Threads for Fine-Grained Time-Sharing

Based on the observed variability in per-loop speedups for these programs, we began to explore lightweight means of assigning threads to processors such that the system can be multiprogrammed efficiently. We describe our thread management mechanism and policy in the subsections that follow. Section 5 then evaluates these ideas and suggests ways of expanding them.

4.1 Adaptive Thread Management: Mechanism

In Section 2, we described that the standard SUIF run-time system relies on a master-worker approach to release worker threads at a parallel loop with information about how to execute the loop. This information includes the location of the loop in the program, the variables accessed by the loop, and the block of iterations to execute. The standard SUIF run-time system is capable of releasing fewer than the full allocation of worker threads to execute a loop, but *any unused threads remain spinning at the barrier.*

Our adaptive run-time system is implemented by making a small change to the standard SUIF mechanism for managing threads. At each loop invocation, the run-time system computes the number of threads to use. Then, the run-time system releases only those threads to the computation; the remaining threads do not spin but *instead are available to be applied to useful work by other processes.*

When multiple programs are run, **max_procs** threads are allocated in total, where **max_procs** is the number of available processors. Each program has its own master thread, and the programs share the remaining threads to use as workers. To avoid memory access overhead for odd and prime partitioning of data, a loop is always allocated an even number of threads. Thus, if there are **N** programs being run on **max_procs** threads, each loop can allocate at most $\texttt{procs\_per\_prog} = \lfloor(\texttt{max\_procs} - \texttt{N})/2\rfloor * 2$ threads. When one of the programs finishes its execution, it releases the thread it used as a master and makes it available to the remaining executing programs.

4.2 Adaptive Thread Management: Policies

The previous subsection focused on implementing a *mechanism* for adaptive thread assignments and abstracted away the related *policy* decisions. Whether a program's per-loop speedup variability is caused by its inherent characteristics, or by external competition for processors amongst programs, this variability makes useful parallelism difficult to estimate at compile-time. Thus, the decision of how many threads to give a program must be made throughout the program's execution. Here we examine policies for repeatedly making this decision at run-time.

A simple adaption policy would base the processor allocation decision for some loop L on its observed speedups in past invocations. For example, at the core of our adaption policy, the per-loop, one-processor time seq_time_L is computed by averaging over all of this loop's invocations in a sequential pre-run of the program.[1] This information is placed in a file and is read in at the beginning of the adaptive run. By default, the program starts by assigning each loop the number of threads originally requested by the application. On subsequent loop invocations, the number of allocated threads can be adapted upwards or downwards to directly match the observed loop speedups. While initially appealing, we found this simple policy to be inadequate at balancing the performance needs of multiple running programs. Thus, additional factors were added, as discussed below.

Delaying Adaption The adaption policy must aim to optimize the processor utilization without unduly penalizing programs for brief, possibly-spurious fluctuations in their performance. Because of the simple block partitioning of iterations used by the compiler, an adaption involves a remapping of iterations to processors, and as a direct result,

[1] The point of the sequential profiling run is to give our prototype sufficient information to gauge loop speedups at run-time. The latency of this profiling run could be considered part of the compilation process. For programs whose sequential run-time is prohibitively high, a two- or four-processor run could be substituted, with the processor adjustments modified appropriately. One could also omit the profiling pre-run entirely by gathering sequential profiling information on the first several invocations of each loop in the "real" run, before going to normal parallel execution for the remainder of the run.

a repartitioning of data accessed by each processor. Thus, at least the first invocation of the loop at a new number of threads may observe poor performance due to increased memory costs. For this reason, we typically do not want to adapt downwards at the first sign of poor speedup, but instead should delay this decision to see if it is a sustained event.

Our first tactic is to require that the speedup change by a significant factor before adaption is considered. That is, if the previous speedup is above $factor_{up}$ of ideal, the number of threads may be adapted up. If it is less than $factor_{down}$ of ideal, the number of threads may be adapted down. (To improve performance, adapting from two processors to one processor is treated as a special case with a slightly larger factor used.)

Speedup changes that exceed the parameterized factors cause the loop to be considered for adaption, but the system does not immediately adapt down due to one bad timing. Instead, the system must observe more than *wait* bad timings in a row before adapting down. This allows applications to have occasionally-anomalous behavior without punishment. Our results indicate that a non-zero *wait* is particularly important immediately following an adaption; we observed up to a 9% performance improvement for the combined programs by waiting beyond a single bad timing.

Competition for Threads If several programs with good speedups are running concurrently, then sometimes a loop may be allocated more threads for its next invocation than the system currently has available. In these cases, the loop suspends execution in a FIFO queue until the threads are available. The time spent waiting in the queue is part of the monitored execution time, so that if programs are competing with each other for threads, the system will naturally adapt down the number of threads being used by all programs. Ultimately this allocates more threads to the loops that are achieving the best speedups. As previously stated, the adaption mechanism is gradual, adapting up or down only 2 threads at a time. This lets one program slowly increase its resource allocation without suddenly impacting the performance of other executing programs. Our experiments have indicated that this gradual change significantly improves performance over selecting the number of processors more directly according to observed speedups; the system achieved up to a 33% performance improvement by introducing gradual downward adaption.

Sequentializing Loops An important decision in automatic parallelization is determining when to sequentialize a loop's execution. When a loop is sequentialized, it will never be executed in parallel again for the remainder of the program's current execution. Sequentializing a loop eliminates the normal parallelization overhead, and is thus an important efficiency feature for fine-grained, low-speedup loops. In SUIF and other current parallelizing compilers, loops are sequentialized at *compile-time* via potentially-inaccurate static performance estimates and user-defined threshold values. This can lead to poor performance because compile-time estimates can frequently be inaccurate. Our adaption policy sequentializes loops based on *observed run-time performance*; this allows more accurate assessments of their potential for parallelism. Run-time sequentialization is an important efficiency feature, but it must be used conservatively so as to avoid sequentializing loops that could profitably be executed in parallel. We set a parameter, *granularity_threshold*, which is a per-invocation minimum execution time for profitable execution in parallel.

Based on the observations and data discussed above, we arrived at the adaption policy presented in Figure 2. For the experiments presented in this paper, we use the following values for these parameters: $granularity\_threshold = .001s$, $factor_{up} = .67$, $factor_{down} = .5$, and $wait = 1$.

5 Experimental Results

Adaptive runtime parallelism has two means by which it might improve system performance. First, in some applications individual program speedups may degrade (primarily due to insufficient granularity and false sharing) if too many processors are used. For the applications and number of processors we studied, however, we found this to be a fairly minor effect. Rather, the real advantage of adapting the number of processors applied to a loop is that the system can increase throughput and processor utilization by matching the loop to the number of processors it can effectively exploit and releasing remaining threads for useful work in other programs. In this section, we present a series of experimental results showing increased overall throughput when using the adaptive run-time system to execute multiple programs at a time.

Initialization.

if $seq\_time_L < granularity\_threshold$ then
  sequentialize L
else
  $procs_L(1) = \min(procs\_per\_prog, iters_L)$

General adaption decision.

if $speedup_L(k-1) > procs_L(k-1) * factor_{up}$ then
  $procs_L(k) = \min(procs_L(k-1)+2, procs\_per\_prog, iters_L(k))$
  reset wait
else if $speedup_L(k-1) < procs_L(k-1) * factor_{down}$ then
  wait = wait - 1
  if (wait < 0) then
    $procs_L(k) = \min(procs_L(k-1)-2, iters_L(k))$
    reset wait
    if ($procs_L(k) \leq 1$) then
      sequentialize L
else
  $procs_L(k) = min(procs_L(k-1), iters_L(k))$
  reset wait

Figure 2: Pseudo-code for the implemented process adaption policy.

5.1 Methodology

Evaluating our proposal requires a mechanism for controlling threads and assigning them to different tasks in a multiprogrammed environment. We originally implemented this functionality using the available blocking system calls on the SGI Power Challenge. We found, however, that they were prohibitively expensive, typically resulting in a 2X or more *slowdown* over the original codes. For these reasons, we developed an alternative methodology for our experiments. We extended the run-time system to support executing multiple programs together. We compiled and linked the multiple programs into a single executable, accordingly modifying where the individual programs received their input and placed their output to avoid conflicts. Thus, the run-time system could control the schedule of a set of `max_procs` threads and how they were used by a group of programs. Obviously, in a production system, we would not require compiling the programs together but would instead have them link in a shared library and execute using a common pool of threads. System support for this approach should be considered in future developments. We anticipate similar overheads with either approach, but the methodology we chose was simpler for a proof-of-concept implementation.

5.2 Adaption Measurements

We measured the effectiveness of the adaptive run-time system on combinations of the 5 programs described in Section 3. Table 5.2 presents the 8, 12 and 14-processor results, showing the speedup of the combined program compared to the sum of the individual programs' execution times each running on a *standalone machine*. (A `P`-processor run is one in which the sum total of processors, or threads, used by both programs is `P`.) Consider for example, the entry in the 14-processor section corresponding to hydro2d-cgm. This entry contains the sum of each of cgm's and hydro2d's individual standalone 14-processor execution times, 66.0s and 44.7s, divided by the combined, thread-managed 14-processor execution time of 83.5s. Numbers greater than 1.0 indicate that our adaptive thread management policy improved the workload's performance.[2]

In fact, we find that our adaptive thread-management policy almost always improves performance. On all the runs shown, only 5 of the 29 runs fail to be improved by thread-management when compared to standalone execution. When

[2] The asterisk by the 12-processor swim results indicate that these programs were run using a coarser granularity timer than the other combinations. Our experiments showed consistently better results with the finer granularity timer (see discussion), so these results should be considered pessimistic compared to the better timer.

8-Processor Runs

	Cgm	Hydro2d	Mgrid	Su2cor
Hydro2d	1.18	–	–	–
Mgrid	1.32	.97	–	–
Su2cor	1.20	1.17	1.03	–
Swim	1.33	.99	0.78	1.11

12-Processor Runs

	Cgm	Hydro2d	Mgrid	Su2cor
Hydro2d	1.23	–	–	–
Mgrid	1.14	1.09	–	–
Su2cor	1.24	1.23	1.03	–
Swim	1.19*	1.08*	.84*	n/a

14-Processor Runs

	Cgm	Hydro2d	Mgrid	Su2cor
Hydro2d	1.33	–	–	–
Mgrid	1.19	1.18	–	–
Su2cor	1.31	1.22	1.16	–
Swim	1.14	1.15	0.90	1.04

Table II: Speedup factors comparing execution times of thread-managed application workloads to unmanaged execution (8-, 12-, and 14-processor runs).

one considers only runs with 12 or more processors, the results are even more promising: all but one workload show improvements due to our adaptive thread-management policy. This is because individual programs tend to use higher-numbers of processors less efficiently; thus there is more leverage for thread-management to redistribute resources and improve performance. In some cases, our approach yields up to 1.33X speedups over executing the programs one at a time in standalone mode. Note further that comparing against standalone execution is a fairly conservative evaluation metric for our approach. Rarely are these machines *available* to users in standalone mode! More realistically, the programs are run as part of a multiprogrammed workload, where an individual program loses parallelism efficiency due to competition for resources with other programs, as occurs in our experiments when executing programs together. Thus, our approach should be considered an alternative to multiprogrammed runs without thread management, and not standalone runs. We believe adaptive thread-management would show even more significant improvements if compared against more realistic coarser-grained scheduling approaches such as gang-scheduling. This is especially true for programs where parallel speedups are fairly high, and coarser-grained approaches fail to respond to their needs fast enough.

Examining the individual results, we observe a number of trends. At 8 processors, most combinations improve. Improvements range from roughly 11% for swim-su2cor to 33% for some of the workloads involving cgm. On the other hand, there are three workloads where performance does not improve; these include almost negligible degradations in the mgrid-hydro2d and swim-hydro2d combinations, and a larger degradation with the mgrid-swim combination. It is not surprising to see some degradation when combining mgrid and swim, since these are two coarse-grain programs that speed up extremely well when run individually. Thus, the workloads that involve them can be challenging because each application has such good individual speedups that it offers little leverage for adaption-based improvements.

As more processors are considered, adaptive thread-management becomes a clear winner. At 12 to 14 processors, the individual applications running on a standalone system often do not make efficient use of all the available parallelism. While nearly all the programs contain loops that speed up well to 14 processors, the per-loop speedups are generally fairly variable, and more of the execution time is spent on the less-efficient loops. This gives adaptive thread-management better opportunities to overlap execution with other programs without paying a high performance penalty. One way to visualize these opportunities is in terms of where each application is positioned on its individual speedup curve. As the available number of processors is increased (i.e., as one moves out along the speedup curve) a program is increasingly unlikely to be able to use them all efficiently. In these cases, downward adaption can move back down along the speedup curve, toward a more efficient region of the curve; this helps overall workload performance.

In particular, in the 12 and 14-processor runs, only the mgrid-swim workload continues to show a degradation.

The remaining applications improve between 3 and 33%. Even the challenging mgrid-swim workload, however, shows better behavior in the 14-processor runs. While overall the results improve as number of processors increase, a few workloads involving cgm and su2cor are less improved at 12 or 14 than they were at 8 processors. This effect is apparently caused by significant low-parallelism components to these programs; when run with other programs that speed up well, the execution times of cgm and su2cor start to dominate overall execution times.

5.3 Discussion

The presented results offer a view of the opportunities and challenges presented by dynamically managing parallelism in compiler-parallelized codes. As a result of this work, we identify a few implementation issues that bear discussion.

First, we should touch on the envisioned user-model for our adaptive threads approach. Clearly we do not expect programmers to compile programs together as we did for expediency in this experiment. Rather, the environment will allow for individual programs to be compiled and then executed by attaching to the run-time system and thread manager. The key attribute of any such system is that there be a neutral party in charge of managing threads, and that there be a lightweight mechanism for allocating or relinquishing threads such that an application pays little or no cost for blocking a thread as compared with leaving it spinning in a synchronization loop. For example, it would be possible to build such a system in which parallel programs are invoked similarly to dynamically shared libraries on current systems. Since we foresee an environment in which the bulk of the applications are uniprocessor or compiler-parallelized multiprocessor applications, the compiler can add in the appropriate run-time system calls to ensure that the user-level processes on the system subscribe to the thread manager, and yet the users need not be aware of the difference.

Our second observation pertains to timer granularities. Our adaption policies rely on loop timings accurate at the same granularity as loops that can profitably execute in parallel. Inaccurate measures of loop execution times may impede decisions on how many processors a loop should be given. System developers should provide fine-grained, *user-readable* timers, so that performance-aware software can time at sufficiently fine granularities.

6 Related Work

This section will present some of the prior work on multiprocessor scheduling on which our work extends and discuss some relevant details of our work.

Some related work, such as that by Setia et al. [8] and Majumdar et al. [5] has primarily explored analytical mechanisms for evaluating the efficiency of different scheduling approaches. Building on this analytical work, several more empirical studies have explored thread management in real systems [6, 14, 15, 2]. In some of these studies, such as [6], the thread management policy assumed that each running program was efficiently using all of the threads it had requested; no on-line performance measurement was used to doublecheck on the parallel efficiency of the running programs. This gives users an undue temptation to request as many threads as possible, even if the program will not use them effectively. Other studies, including those by Tucker et al. [14, 15], base thread allocation on specific synchronization mechanisms in the code, and thus require that the programmer of the system use these synchronization mechanisms as the basis of an explicit parallel programming model.

One particular scheduling policy, gang scheduling, has received a great deal of attention. In gang scheduling, processes working on behalf of the same program are scheduled to run together where possible, in order to reap benefits of faster communication and improved memory behavior. Both analytical [11] and experimental studies [10, 9, 4] have been published. Our scheduling policy is a good deal more sophisticated than the heuristic-based gang scheduling, however, since it bases processor allocations on aggregate observed performance, rather than on heuristics regarding what "should" produce good performance.

One recent study particularly warrants a close comparison here. Unlike prior papers which focused on explicitly-parallelized codes, Yue and Lilja have performed scheduling studies on compiler-parallelized applications [17]. Like ours, their LLPC technique also adapts the number of threads allocated to a parallel loop dynamically, but in their case the decision is based on just system load and a static estimate of work in the loop. A key contribution of our work, as compared both to the fully-static SUIF approach and the partially-static LLPC approach, is that we use a fully-dynamic runtime performance measure to guide adaption, rather than rough static guesses. A direct comparison with LLPC is impossible, but we discuss a few key points here. Their paper shows consistent improvements over SGI's gang scheduling technique, and improves over space sharing techniques for the programs that speed up well. It is important to note, however, that had they applied our speedup metric (from Table 5.2) to their results, their "speedups" for executing two programs together would all be less than 0.9, in some cases by a significant margin. (This assertion is based on comparing estimates from their graphs and reported execution times.) For this reason, we argue that the adaptive thread-management policy we demonstrate here shows convincing new improvements over standalone individual execution times and suggests significant opportunities to improve multiprogramming effectiveness.

7 Conclusions

Overall, our results highlight the promise of using on-the-fly monitoring in managing threads for compiler-parallelized workloads. Our technique improves the parallel efficiency of workloads by adjusting thread assignments based on the achieved speedups of the programs being executed. This approach has demonstrated substantial improvements, in several cases executing 33% faster than the individual programs executed on a *standalone* machine. These results improve on those garnered from established scheduling regimes like gang scheduling, and also with other research approaches [17].

Our experimentation with different thread control policies has demonstrated the importance of (i) making the adaption mechanism gradual, (ii) increasing or decreasing the number of threads allocated to a computation only a few processors per invocation, and (iii) allowing for spurious timings due to changes in processor allocation. Gradual adaption is particularly important for allowing the memory access behavior to stabilize and to avoid significant conflict with other executing programs; intuitively, this issue arises because the same loop often executes repeatedly in quick succession, so a computation may be competing with the same computation in other executing applications for an interval of time.

Taking the three major policy issues into account, we have shown that the benefits of adaption increase as the number of processors increases because individual compiler-parallelized applications are less able to make efficient use of all the available processors. Based on this observation, we believe our techniques will become increasingly relevant as multiprocessor sizes increase. Scheduling mechanisms such as ours will be vital in the near future when it will be routine to have multiprogrammed workloads including compiler-parallelized applications executing on medium and large scale parallel machines.

Acknowledgements. The authors wish to thank Kimo Yap for his multiprogrammed extensions to the SUIF run-time system upon which this work is based. Mary Hall is supported by DARPA contract DABT63-95-C0118. Margaret Martonosi is supported in part by an NSF Experimental Systems Program grant (MIP-9708624) and by an NSF CAREER award (CCR-0502516).

References

[1] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12), December 1996.

[2] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proc. ACM Sigmetrics Conference on Measurement and Monitoring of Computer Systems*, May 1991.

[3] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S. Liao, E. Bugnion, and M.S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12), December 1996.

[4] A. Hori, H. Tezuka, Y. Ishikawa, N. Soda, H. Konaka, and M. Maeda. Implementation of gang-scheduling on workstation cluster. In *Proc. 10th International Parallel Processing Symposium*, April 1996.

[5] S. Majumdar, D.L. Eager, and R.B. Bunt. Scheduling in multiprogrammed parallel systems. In *1988 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, May 1988.

[6] 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.

[7] W. Saphir, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.1 Results. Technical Report NAS-96-010, NASA Ames Research Center, August 1996.

[8] S.K. Setia, M.S. Squillante, and S.K Tripathi. Processor scheduling on multiprogrammed, distributed memory parallel computers. In *1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, May 1993.

[9] C. Severance and R. Enbody. Automatic self-allocating threads (ASAT) on an SGI Challenge. In *Proc. 1996 International Conference on Parallel Processing*, August 1996.

[10] C. Severance, R. Enbody, S. Wallach, and B. Funkhouser. Automatic self-allocating threads (ASAT) on the Convex Exemplar. In *Proc. 1995 International Conference on Parallel Processing*, August 1995.

[11] M.S. Squillante, F. Wang, and M. Papaefthymiou. An analysis of gang scheduling for multiprogrammed parallel computing environments. In *Proceedings of Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[12] The Standard Performance Evaluation Corporation. World Wide Web Site. http://www.specbench.org, December 1996.

[13] Evan Torrie, Margaret Martonosi, Chau-Wen Tseng, and Mary W. Hall. Characterizing the Memory Behavior of Compiler-Parallelized Applications. *IEEE Trans. on Parallel and Distributed Systems*, December 1996.

[14] Andrew Tucker. *Efficient Scheduling on Multiprogrammed Shared-Memory Multiprocessors*. PhD thesis, Stanford University, December 1993. Also Computer Systems Laboratory Tech. Report CSL-TR-94-601.

[15] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proc. 12th ACM Symposium on Operating System Principles*, October 1991.

[16] R. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[17] K.K. Yue and D.J. Lilja. Efficient execution of parallel applications in multiprogrammed multiprocessor systems. In *Proc. 10th International Parallel Processing Symposium*, April 1996.