Efficient Multiprogramming for Multicores with SCAF

Timothy Creech   Aparna Kotha   Rajeev Barua

Department of Electrical and Computer Engineering
University of Maryland, College Park
College Park, MD 20742
{tcreech,akotha,barua}@umd.edu

ABSTRACT

As hardware becomes increasingly parallel and the availability of scalable parallel software improves, the problem of managing multiple multithreaded applications (processes) becomes important. *Malleable* processes, which can vary the number of threads used as they run, enable sophisticated and flexible resource management. Although many existing applications parallelized for SMPs with parallel runtimes are in fact already malleable, deployed run-time environments provide no interface nor any strategy for intelligently allocating hardware threads or even preventing oversubscription. Work up until SCAF either depends upon profiling applications ahead of time in order to make good decisions about allocations, or does not account for process efficiency at all. This paper presents the **Sc**heduling and **A**llocation with **F**eedback (SCAF) system, a drop-in runtime solution which supports existing malleable applications in making intelligent allocation decisions based on observed efficiency without any paradigm change, changes to semantics, program modification, offline profiling, or even recompilation. Our existing implementation can control most unmodified OpenMP applications. Other malleable threading libraries can also easily be supported with small modifications, without requiring application modification.

In this work, we present the SCAF daemon and a SCAF-aware port of the GNU OpenMP runtime. We demonstrate that applications running on the SCAF runtime still perform well when executing on a quiescent system. We present a new technique for estimating process efficiency purely at runtime using available hardware counters, and demonstrate its effectiveness in aiding allocation decisions.

We evaluated SCAF using NAS NPB parallel benchmarks. When run concurrently pairwise, 70% of benchmark pairs on an 8-core Xeon processor saw improvements averaging 15% in sum of speedups compared to equipartitioning. For a 64-context Sparc T2 processor, 57% of pairs saw a similar 15% improvement. The improvement was 45% vs. equipartitioning when three selected benchmarks were concurrently run.

MICRO-46 December 7–11, 2013, Davis, CA, USA
Copyright 2013 ACM 978-1-4503-2638-4/13/12 ...$15.00.
http://dx.doi.org/10.1145/2540708.2540737

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*Multiprocessing/multiprogramming/multitasking, Threads*; D.3.4 [**Programming Languages**]: Processors—*Run-time environments*

General Terms

Management, Measurement, Performance

Keywords

Multithreaded programming, oversubscription, parallelism, resource management, user-level scheduling

1. INTRODUCTION

When running multiple parallelized programs on many-core systems, such as Tilera's TilePro64, or even large Intel x86-based SMP systems, a problem becomes apparent: each application makes the assumption that it is the only application running, and consequently the machine is quickly oversubscribed if multiple programs are running. A machine is said to be oversubscribed when the number of computationally intensive threads exceeds the number of available hardware contexts. This problem becomes increasingly important as we move to systems with more cores than applications, where space sharing of cores between applications is desirable but rarely done in practice. Modern operating systems attempt to time-share the hardware resources by context switching, but this is a poor solution. The context switching incurs additional overhead. Further, when some threads participating in a barrier are context-switched out, other threads in the same barrier may incur long waiting times, reducing system throughput. Finally, some synchronization techniques, such as spinlocks, depend heavily on the presence of dedicated hardware contexts for reasonable performance.

In this paper, we define the parallel efficiency of executed code to be $E \equiv \frac{S}{p}$, where executing the code in parallel on p hardware contexts yielded a speedup S over serial execution. We say that a multiprogrammed parallel system has high total efficiency when the sum of speedups achieved by its processes is high. That is, the average hardware context contributes a large speedup.

When dealing with truly malleable parallel programs, the ideal solution is to actually change the number of software threads that need to be scheduled, approximating space sharing. We argue that this should be done automatically and transparently to the system's users. Existing parallelization runtimes generally allow users to specify the maximum number of threads to be used at compile-time or run-time. However, this number remains fixed for the duration of the program's execution, and it is unreasonable to expect users on a system to manually coordinate. Furthermore, even on a single-user system it is not always clear how best to allocate hardware contexts between applications if good total system efficiency is desired: scalability may vary per application and across inputs. Finally, in order for a solution to be adopted, it should not require new programming paradigms or place excessive demands on the users. That is, users should not be required to rewrite, recompile, profile, or spend time carefully characterizing their programs in order to benefit. We believe that this cost to the user is the primary reason that none of the literature's existing solutions have enjoyed widespread adoption.

As a result, we sought to create a scheme which satisfies the following requirements:

- Total system efficiency is optimized, taking individual processes' parallel efficiencies into account
- No setup, tuning, or manual initialization of any participating application is required before runtime
- No modification to, or recompilation of, any participating application is required
- Effectiveness in both batch and real-time processing scenarios
- System load resulting from both parallel processes which are not truly malleable, as well as processes which are not participating in the SCAF system, is taken into account

Looking at existing research and experiments, the ingredients seem to be available. Some solutions gather information on efficiency by making use of pre-execution profiling[7, 12], while others do not require profiling and do not account for program efficiency[13, 15, 8, 18]. However, it is nontrivial to measure and account for program efficiency *without* profiling. This task is made more difficult by the fact that we want to avoid modifying or even recompiling any applications – instrumentation and communication between SCAF processes must be added only as modifications to the compiler's parallelization runtime libraries.

SCAF solves this problem with a technique which allows a client to estimate its *efficiency* entirely at runtime, alleviating the need for pre-execution profiling. To understand how, consider that parallel efficiency can only be measured by knowing speedup vs serial execution. However, in a parallel program there is no serial equivalent of parallel sections; hence serial measurements are not directly available. Running the entire parallel section in serial is possible, but can greatly slow down execution, overwhelming any benefits from malleability.

We solve the problem of measuring serial performance at low cost by cloning the parallel process into a serial experimental process the first time each parallel section is encountered dynamically. During this first run, the serial experiment is run concurrently with the parallel code as long as the latter executes. The parallel process runs on $N-1$ cores, and the serial process on 1 core, where N is the number of threads the parallel process has been allocated. Crucially, the serial thread is not run to completion (which would be expensive in run-time overhead); instead it is run for the same time as the current parallel section. We find this gives a good enough estimate of serial performance to be able to estimate efficiency. *The run-time overhead of the serial experiment is very low because it is run only once during the first dynamic instance of the parallel section. Subsequent dynamic runs of the parallel section run on N cores without running the serial experiment.*

We demonstrate that the SCAF runtime incurs low overhead on a process running alone, because experiments are infrequent. Furthermore, we evaluate the SCAF system in various multiprogramming scenarios, showing that SCAF generally improves system efficiency over both an unmodified OpenMP runtime and simple equipartitioning.

We plan to make SCAF open-source by the time of publication. An executable version will be available for users. A source-code version will be available to aid researchers.

2. RELATED WORK

SCAF seeks to solve performance and administrative problems related to the execution of multiple multithreaded applications on a many-core shared-memory system. This section outlines related work, both in the world of shared-memory parallelism, and in the world of distributed-memory parallelism. The section is concluded with a concise table (Table 1) of features and properties satisfied by the various systems and SCAF.

2.1 Distributed Memory Systems

Flexible and dynamic scheduling for distributed memory parallel applications and systems has been an active area of research. SCAF does not compete in the distributed memory world, as it is designed to solve problems pertaining to shared-memory systems. However, since the problems are similar at a high level, we briefly describe some of the related work in this section.

Kale et al [11] implemented a system for dynamically reconfiguring MPI-based applications through a system using a processor virtualization layer. Crucially, this allows the migration of work from one node of the distributed system to another. Load balancing is effectively achieved by creating many virtual processes for each physical processor. The system then can reconfigure parallel jobs at runtime based on the arrival or departure of other jobs. However, recompilation of a participating application is required, and small modifications to the source code are necessary.

Sudarsan et al [16] improved on this work with ReSHAPE, their framework for dynamic resizing and scheduling. Using the provided resizing library and API, application users can specify shared variables suitable for redistribution between iterations of an outer loop. The points at which redistribution is safe must be specified by the programmer. Between each iteration, a runtime scheduler makes decisions on whether to expand or shrink a job based on node availability and observed whole-application performance. The primary disadvantage of ReSHAPE is that it requires applications to be significantly rewritten to use their API.

2.2 Shared Memory Systems

Relatively little work has been done concerning multiprogrammed, multithreaded scheduling and the problem of

Implementation	Avoids recompilation	Avoids modifications	Considers process efficiency	Avoids a priori testing / setup	Multi-process support
SCAF	✓	✓	✓	✓	✓
RSM [13]	×	×	×	✓	✓
DTiO [15]	✓	✓	×	(N/A)	✓
ERMfMA [8]	×	×	×	(N/A)	✓
APiCPC [7]	×	✓	✓	×	✓
Hood [3]	×	×	×	(N/A)	✓
PCfMSMP [18]	✓	✓	×	(N/A)	✓
Lithe [14]	✓	✓	×	(N/A)	×
CDPAS [12]	×	×	✓	×	✓

Table 1: Feature comparison of related implementations (ad hoc acronyms used for brevity)

oversubscription. Tucker et al [18] observed serious performance degradation in the face of oversubscription on shared-memory multiprocessors. They showed that by modifying a version of the Brown University Threads package used on an Encore Multimax, a centralized daemon can (strictly) limit the number of running threads on the system to avoid oversubscription by suspending threads when necessary. By modifying only the system's threads package, they were able to support many programs using that threads package without modification. However, their work has several disadvantages as compared to SCAF work: (1) the partitioning policy does not take into account any runtime performance measurements, but assumes all processes are scaling equally well, and (2), the scheme's ability to control the running number of threads depends on the use of the specific parallel paradigm where the programmer creates a queue of tasks to be executed by the threads, and the assumption that the application does not depend on having a certain number of threads running. If an application does not meet both requirements, then it may run incorrectly without warning. This is a restriction of operating within a threads package where unsupported program behavior cannot always be detected at runtime. By contrast, SCAF offers modified runtime libraries which provide higher-level abstractions. Unsupported program behavior which would imply non-malleability is detected as it is requested, after which SCAF avoids incorrect behavior by holding the number of threads fixed for that running program.

Arora et al [2] designed a strictly user-level, work-stealing thread scheduler which was implemented in the "Hood" [3] prototype C++ threads library, and later in the Cilk-5 [6] language's runtime system. Work stealing is an approach in which the programmer specifies all available parallelism in a declarative manner, and then the implementation schedules parallel work to a certain number of worker threads (using deques, or simply work stealing queues) which are allowed to "steal" work from one another in order to load balance. The number of worker threads is usually equal to the number of available hardware contexts. The problem that Arora solves is that when multiple processes are running, each doing work stealing with multithreading, then having independent worker threads for each process leads to more worker threads than hardware contexts, leading to over-subscription of the machine and poor performance. Their approach is to combine the work-stealing queues across applications, and using a number of shared workers that does not result in oversubscription. Their approach also accounts for serial processes when avoiding oversubscription.

The approach used in Hood[3] has several differences with the goals and capabilities of SCAF. First, Hood can only reduce oversubscription when the parallel processes all utilize work-stealing libraries. In contrast, SCAF reduces oversubscription for any malleable parallel processes, regardless of whether they use work stealing or not. Second, although Hood and SCAF have the same goal of avoiding oversubscription, they do so using different mechanisms: SCAF relies on malleable processes to reduce the number of threads, whereas Hood has a specialized solution for work stealing that relies on the work stealing programming model, without taking advantage of malleability. Third, parallel threads using Hood are not allowed to use blocking synchronization, since Hood might swap out a lock-holding thread, which would prevent other threads from making progress. SCAF has no such restriction, since it reduces the number of threads created, rather than allowing threads to be swapped out. Fourth, the implementation of Hood is complex and difficult since it has the same restriction as user processes, in that Hood code must not use blocking synchronization, which is simpler, but might cause tremendous slowdowns if the kernel preempts a process which holds locks, causing other workers to block. Fifth, Hood does not take any run-time measurements, and hence cannot favor processes with better scalability, whereas SCAF does, which helps to improve overall system throughput by rewarding processes that scale better.

Hall et al [7] performed experiments that emulate using a similar centralized daemon and modifications to the Stanford SUIF auto-parallelizing compiler to dynamically increase or decrease the number of threads at the start of a parallel section based on system load and runtime measurements of how effectively each parallel section uses its hardware contexts. Kazi et al [12] adapted four parallel Java applications to their own parallelization model and implementation so that each application reacts to observed system load and runtime performance measurements in order to increase or decrease its number of threads at runtime before each parallel section. SCAF builds on ideas developed in these works. Compared to SCAF, their systems have the following drawbacks: (1) they require recompilation or modification of the programs in order to control the number of threads; and (2) despite controlling compilation, they are unable to avoid depending on a priori profiling for making allocation decisions. SCAF works with unmodified, SCAF-oblivious binaries, and collects all of its information regarding efficiency during program execution, avoiding the need for careful application profiling.

Suleman et al [17] describes a feedback-based system for choosing the optimal number of threads for a single program at runtime. Specifically, the system can decrease the number of threads used in order to improve efficiency in the face of critical sections and bus saturation. This system requires no a priori knowledge of programs, and utilizes a serialized "training" phase to reason about serial and parallel performance. However, the system does not attempt nor claim to reason about multiprogramming, and it is unclear if it could be adapted to do so. SCAF carefully avoids any serialization and seeks primarily to handle multiprogramming.

More recently, Pan et al [14] created the "Lithe" system for preventing hardware oversubscription within a single application, or process, which composes multiple parallel libraries. This is a separate problem from the one discussed in this paper. For example, consider a single OpenMP-parallelized application which makes a call to an Intel TBB-parallelized library function. The result is often significant oversubscription: on a system with N hardware contexts, the OpenMP parallel section will allocate N threads, and then each of those threads will create another N threads when Intel TBB is invoked, resulting in N^2 threads. The Lithe system transparently supports this composition in existing OpenMP and/or Intel TBB binaries by providing a set of Lithe-aware dynamically-loaded shared libraries. However, it should be made clear that Lithe makes no attempt to coordinate multiple *applications* running concurrently, and does not vary the number of threads which the application is using at runtime. Lithe strictly avoids oversubscription potentially resulting from composition of parallel libraries within a single process. SCAF builds on Lithe's idea of supporting existing applications via modified runtime libraries, but focuses instead on the composition of parallel libraries used in separate, concurrently-executing executables.

McFarland [13] created a prototype system called "RSM," which includes a programming API and accompanying runtime system for OpenMP applications. The application must be modified to communicate with the runtime system via API calls between parallel sections. Once recompiled, the application communicates with the RSM daemon and depends upon it for decisions regarding the number of threads to load beginning with the next parallel section. The RSM daemon attempts to make allocation decisions according to observations of how much work is being performed by each process at runtime. An application's useful work is taken to be the number of instructions retired per thread-seconds. Processes are given larger allocations if they perform more useful work. Unlike RSM, SCAF does not require program recompilation. Further, SCAF compares *efficiency* observed at runtime, considering the *improvement* in IPC[1] gained by parallelization, whereas RSM only considers the *absolute* IPC of each process.

In the interest of preserving existing standards and interfaces, Schonherr et al [15] modified GCC's implementation of OpenMP in order to prevent oversubscription. The implementation supports applications without recompilation. However, their system implements only a simple "fair" allocation policy, where all applications are assumed to scale equally well, and no runtime performance information is taken into account.

Hungershöfer et al [8] implements a runtime system and daemon for avoiding oversubscription in SMP-parallel applications. Their system requires modifications to the applications involved, and provides a centralized server process which controls thread allocation. However, their method for maximizing accumulated speedup depends on significant offline analysis of the applications for determining their speedup behaviors, parallel runtime components, and management/ communication overheads.

2.3 Related Implementations

As part of related work, some solutions have been implemented and explored. These are listed below in order of their similarity to SCAF, and their features are enumerated in Table 1.

1. RSM, [13]
2. Dynamic Teams in OpenMP, [15]
3. Efficient Resource Management for Malleable Applications, [8]
4. Adapting parallelism in compiler-parallelized code, [7]
5. Hood, [3]
6. Process control and scheduling issues for multiprogrammed shared-memory processors, [18]
7. Lithe, [14]
8. A comprehensive dynamic processor allocation scheme for multiprogrammed multiprocessor systems, [12]

3. DESIGN

3.1 System Overview

A system running SCAF consists of any number of malleable processes, any number of non-malleable processes, and the central SCAF daemon. The SCAF daemon is started once, and one instance serves all users on the system. All processes are SCAF-oblivious, and are started by users in the usual uncoordinated fashion. Parallel binaries load SCAF-compliant runtime libraries in place of the unmodified runtime libraries — this does not require program modification or recompilation. The SCAF-compliant libraries automatically determine whether a process is malleable at runtime, transparently to the user. A non-malleable process will proceed as usual, requiring no communication with the SCAF daemon, while a malleable process will consult with the SCAF daemon throughout its execution. A process which loads no SCAF-compliant parallel runtime libraries is assumed to be non-malleable and proceeds normally. The SCAF daemon is responsible for accounting for the load incurred by any non-malleable processes. Specifically, hardware contexts used by non-malleable processes must be considered unavailable to malleable processes.

[1] Instructions per cycle

3.2 Conversion from Time-sharing to Space-sharing

By default, modern multi-user operating systems support the execution of multiple multithreaded applications by simple time-sharing. Parallel processes are unaware of one another, and each assume that the entire set of hardware contexts is available. In general, this results in poor efficiency and performance unless the system is otherwise quiescent (*i.e.*, unloaded). As an extreme example, we found that on a small 4-core Intel i5 2500k system running Linux 3.0, the per-instance slowdown when running two instances of the NAS NPB "LU" benchmark (each on 4 threads) was as much as 8X when compared to using space sharing. With the same hardware running FreeBSD 9.0 the penalty was much greater, exhibiting a slowdown of more than 100X. An investigation revealed that LU implements spinlocks in userland which perform poorly without dedicated hardware contexts[1]. Modifying the synchronization primitives used by the system's `libgomp` runtime library won't help, since the problematic synchronization lies in LU itself. Short of perhaps modifying the application, the best solution is space sharing.

The objective of the SCAF system is essentially to effect space-sharing among all hardware contexts running on the system, such that the operating system can schedule active threads to idle hardware contexts and avoid the fine-grained context-switching and load imbalances incurred by heavy time-sharing.

3.3 Sharing Policies

In order to justify the policies which the SCAF daemon implements, a brief discussion of possible policies is useful. The following terminology is used:

- Runtime of a process j: T_j
- Speedup of a process j: S_j
- Threads allocated to process j: p_j
- Number of hardware contexts available: N
- Number of processes running: k

Additionally, we define per-process "efficiency" as $E \equiv \frac{S_j}{p_j}$.

Minimizing the "Make Span"

In distributed memory systems, where users generally submit explicit jobs to a space-sharing job manager, the de facto goal is to minimize the "make span," which is the amount of time required to complete all jobs.

However, the algorithms to solve this problem require precise information concerning not only the speedup behavior of each job, but also accurate estimates of the total work required until a job's completion. This implies a batch-processing model, possible for large distributed memory machines. On shared-memory systems, jobs are run without prior intimation by the user, so the run-time system cannot predict when applications will start, nor when a running application will end. As a result, the make span cannot be applied. Multithreaded processes which operate on a virtually infinite stream of input data are also not uncommon. In this case the "make span" cannot be applied since processes do not necessarily terminate.

Therefore, a new goal is required for a runtime system such as SCAF. Given that the future system load cannot be predicted by the run-time system, we seek an instantaneous metric which will allow SCAF to reason about the performance of processes at runtime. Furthermore, the optimization problem should be constrained such that the system's behavior is consistent with the expectations of an interactive shared-memory machine.

Equipartitioning

When performing equipartitioning, fully "fair" sharing of the hardware resources is achieved, without concern for how efficiently said resources are being used. Each process occupies an equal number of hardware contexts:

$$p_j \leftarrow \left\lfloor \frac{N}{k} \right\rfloor \tag{1}$$

The remaining (N mod k) hardware contexts are distributed arbitrarily among (N mod k) processes to ensure full utilization.

The clear advantage to equipartitioning is simplicity. Oversubscription and underutilization are avoided, and no a-priori performance measurements are required. The problem with equipartitioning is that it can result in low system efficiency. For example, given program A with $S_A(p_A) = 1 + \frac{4}{5}(p_A - 1)$ and program B with $S_B(p_B) = 1 + \frac{1}{8}(p_B - 1)$, one might intuitively want to allow the better-behaved program, A, to use more hardware contexts than B since it makes better use of each hardware context. *However, equipartitioning ignores observed speedup. SCAF does better because it rewards processes with more threads when they are observed to scale well with more threads.*

Maximizing the Sum Speedup

Another appealing approach is to maximize the total sum of speedups achieved by the running processes. That is, given a function $S_j(p_j)$ describing the speedup of each process with p_j threads, maximize $\sum_{\forall j} S_j(p_j)$ by choosing p_j for all $j = 1 \ldots k$. By maximizing the sum speedups, the average speedup obtained per process is maximized. If allocations such as p_j are fixed throughout a program's execution, then this optimization problem only needs to be evaluated one time, when the processes begin execution.

However, this quickly becomes a complex problem with malleable processes where both the speedup function S_p and process allocations can effectively change over time. For example, different parallel sections of code in the same program may vary in how well they make use of hardware contexts. Even if we perform extensive testing and characterization of each parallel section in applications before runtime, in general parallel efficiency may still vary unpredictably due to inputs to the processes. Therefore, what is needed is a system in which efficiency observed only at runtime is taken into account.

Maximizing the Sum Speedup Based on Runtime Feedback

This is the approach used in SCAF. The goal is to partition the available hardware contexts to processes quickly, adjusting over time according to information available at runtime. However, the details of such a system are not immediately clear. When should allocation decisions be made? How do we reason about speedups?

One can begin to imagine a system in which allocation decisions are made per parallel region. However, these parallel regions often begin and end execution at a very high frequency. Hence changing the thread allocation for each parallel region is infeasible since the costs of thread initialization and termination as well as allocation computation would result in prohibitively high overhead. Ideally, the allocation should change relatively infrequently and asynchronously. However it *should* change after longer intervals during an application's run-time, since the application's behavior may change over time, perhaps because it moves to a different phase in the execution. As a corollary, since we cannot possibly react to individual parallel regions, we should reason about speedups in a per-process manner.

Consequent to the discussion above, SCAF clients must maintain and report a single efficiency estimate per process. It is the client's responsibility to distill its efficiency information down to this single constant, and refine it over time. This is a nontrivial task for a pure runtime system since capturing efficiency information requires information on the serial performance of sections. SCAF's lightweight serial experiments, discussed in section 4.3.1, represent a solution for gaining this information without incurring the penalty of temporary serialization, which can be extremely expensive.

By default, lightweight experiments return instructions per cycle (IPC) as measured by the PAPI [9] runtime library since this is generally available from hardware counters. PAPI returns the IPC achieved by the calling process alone. However, experiments could return any metric which is generally indicative of program progress. Floating point operations completed per second (also available from PAPI) may be a more reliable indicator of work if, for example, it is known that the machine is primarily used for floating point work. This is a simple compile-time option in SCAF. However, while it is well understood that IPC is not an ideal work performance metric, we choose to use it by default in SCAF in order to avoid limiting SCAF's usefulness to just floating point workloads.

From a lightweight experiment, SCAF obtains an estimate of the serial IPC of a section. This measurement is then used later at runtime to compare against observed parallel IPC measurements in order to compute the efficiency for that specific parallel section and the process.

The efficiency estimate allows the central SCAF daemon to reason about how efficiently each process makes use of more cores relative to the other clients (processes). Specifically, the daemon uses this efficiency estimate to build a simple speedup model

$$S_j(p_j) \approx 1 + C_j \log p_j, \text{ where } C_j \leftarrow \frac{E_j p'_j - 1}{\log p'_j} \tag{2}$$

where E_j is the reported parallel efficiency from client j, and p'_j is the previous allocation for j. This can be thought of as the simplest form of curve fitting, where the only parameter to the curve is the constant factor C_j. The model describes a simple sublinear speedup curve specified by tuples $(\text{number of threads}, \text{speedup})$ which goes through the points $(1, 1)$ and $(p'_j, E_j p'_j)$, since $S_j = E_j p'_j$. More sophisticated speedup models have certainly been developed [5, 4]. However, SCAF's simple "fitting" is performed repeatedly, adjusting to each round of feedback from the client and reacting dynamically rather than depending on a static model. Such a static model would fail to react to changes in scalability over time, and would require profiling the entire application beforehand.

The above model works well since C_j can be adjusted to approximate speedup curves of real applications using only a single measurement representing recent efficiency. A dynamic system which fits using multiple distinct speedup points might overfit to the application's current behavior, and will react less quickly to changing behavior.

Next, we discuss exactly how the daemon arrives at such allocations. Using the speedup model in equation 2, the SCAF daemon is faced with the optimization problem

$$\max_p \left\{ \sum_{i=1}^{k} S_i(p_i) \middle| p > 0 \bigwedge \sum_{i=1}^{k} p_i = N \right\} \tag{3}$$

or, equivalently using equation 2,

$$\max_p \left\{ \sum_{i=1}^{k} 1 + C_i \log p_i \middle| p > 0 \bigwedge \sum_{i=1}^{k} p_i = N \right\} \tag{4}$$

Let Q_i be defined as $Q_i = \frac{C_i}{\sum_j C_j}$. Since $\sum_j C_j$ and 1 are constant quantities, we can equivalently express our optimization problem as

$$\max_p \left\{ \sum_{i=1}^{k} Q_i \log p_i \middle| p > 0 \bigwedge \sum_{i=1}^{k} p_i = N \right\} \tag{5}$$

Next, we define P_i to be $\frac{p_i}{N}$, such that $\sum_i P_i = 1$. Since $\frac{1}{N}$ is constant, we can express our optimization problem as

$$\max_P \left\{ \sum_{i=1}^{k} Q_i \log P_i \middle| P > 0 \bigwedge \sum_{i=1}^{k} P_i = 1 \right\} \tag{6}$$

We can obtain an equivalent minimization problem by taking the negative of the objective function. Then, we add the constant quantity $\sum_i Q_i \log Q_i$, resulting in

$$\min_P \left\{ \sum_{i=1}^{k} Q_i \log Q_i - \sum_{i=1}^{k} Q_i \log P_i \middle| P > 0 \bigwedge \sum_{i=1}^{k} P_i = 1 \right\} \tag{7}$$

or, after combining sum terms,

$$\min_P \left\{ \sum_{i=1}^{k} Q_i \log \frac{Q_i}{P_i} \middle| P > 0 \bigwedge \sum_{i=1}^{k} P_i = 1 \right\} \tag{8}$$

If we now interpret Q and P as discrete probability distributions, we see that equation 8 describes the relative entropy of of Q with respect to P, or the Kullback-Leibler divergence of P from Q. This relative entropy is known to be always non-negative, and equals zero only if $Q = P$. Therefore, the single optimal solution is at $P = Q$. In other words, $p_i = NQ_i$, or

$$p_i = \frac{NC_i}{\sum_j C_j} \tag{9}$$

As per equation 9, the SCAF daemon sets the allocations p by computing the vector C, then assigning p_i to be the fraction $C_i / \sum C$ of N. Of course, the real allocation needs to be an integer, so p is rounded down, and then any remaining hardware contexts are distributed among the allocations that were rounded down the most. Starvation is avoided by ensuring that no process receives an allocation less than 1.

It can be shown that had our assumed speedup model been linear instead of logarithmic, then the optimization problem becomes immediately uninteresting: the optimal

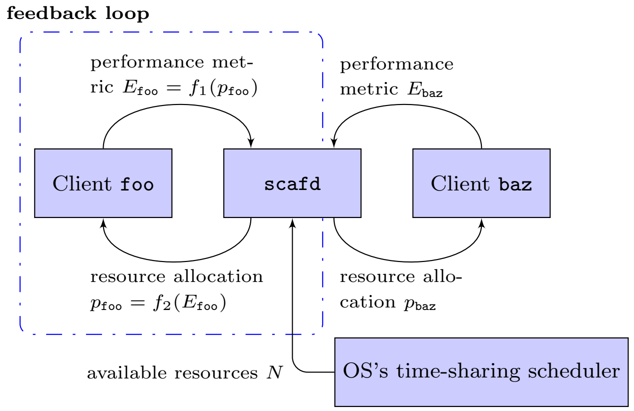

feedback loop
performance metric $E_{\texttt{foo}} = f_1(p_{\texttt{foo}})$
performance metric $E_{\texttt{baz}}$
Client foo
scafd
Client baz
resource allocation $p_{\texttt{foo}} = f_2(E_{\texttt{foo}})$
resource allocation $p_{\texttt{baz}}$
available resources N
OS's time-sharing scheduler

Figure 1: Runtime feedback loop in SCAF's partitioning scheme

solution is always to allocate the entire machine to the single process with the greatest speedup function slope, starving other processes. This would be an undesirable allocation.

Figure 1 illustrates the feedback loop design in SCAF. Consider a machine with 16 hardware contexts. Say process `foo` is observed to scale fairly well, achieving an efficiency of 2/3 on 12 threads, while `baz` is observed scaling poorly, achieving an efficiency of only 3/8 on 4 threads. The SCAF daemon, applying the speedup model and solving the optimization problem, will arrive at $C_{\texttt{foo}} = \frac{8-1}{\log 12}, C_{\texttt{baz}} = \frac{3/2-1}{\log 4}$ and compute new allocations $p_{\texttt{foo}} = \lfloor 14.18 \rfloor = 14, p_{\texttt{baz}} = \lfloor 1.82 \rfloor + 1 = 2$. If the resulting feedback indicates a good match with the predicted models, then the same model and solution will be maintained, and allocations will remain the same. If one or more feedback items indicate a bad fit, either due to a change in program behavior or poor modeling, then a new model will be built using the new feedback information. For example, if `foo` scales better than the $1 + \frac{8-1}{\log 12} \log 14 = 8.43$X speedup anticipated by the previous model, then a new model will be created accordingly, and `foo`'s allocation will increase further.

4. IMPLEMENTATION

4.1 Integrating into Existing Systems

SCAF easily integrates into existing systems without requiring modification or recompilation of programs by providing SCAF-aware versions of parallel runtime libraries. Specifically, our implementation supports OpenMP programs compiled with the GNU Compiler Collection as clients. Just before execution begins, such programs load a shared object which contains the OpenMP runtime, `libgomp`. A user or administrator can specify that the SCAF-aware version of the runtime should be used, making any subsequently launched processes SCAF-aware. SCAF-aware and traditional processes may coexist without issue.

It is worth mentioning that the SCAF-aware implementation of `libgomp` is a one-time modification of the original, involving only 3 lines of changed code and about 2 days of graduate student time. (Mostly spent understanding `libgomp`.) These minor changes call into a `libscaf` client library which is designed to easily support development of additional runtime ports with similar ease. Intel's `libiomp5` has also been ported. Currently, the `libscaf` library itself consists of 622 lines of C.

Although SCAF supports all malleable OpenMP programs, it is important to note that not all OpenMP programs are malleable. Specifically, the OpenMP standard permits programs to request or explicitly set the number of threads in use by the program. Programs that make use of this functionality are assumed by SCAF to be non-malleable, since they may depend on this number. Since SCAF implements the client's OpenMP interface, it can detect when a non-malleable program requests this functionality, and simply consider that application's allocation to be fixed after that point. As a result, SCAF is safe to use as a drop-in replacement for GNU OpenMP on a system even if the system runs a mixture of malleable and non-malleable OpenMP applications.

4.2 The SCAF Daemon

The system-wide SCAF daemon, `scafd`, communicates with the SCAF clients using a portable software bus, namely ZeroMQ [10]. For the sake of portability, the SCAF daemon is implemented entirely in userspace. While the SCAF daemon could run on a separate host, it incurs a small enough load that this is not necessary. The SCAF daemon has three jobs: 1) monitor load on hardware contexts due to uncontrollable processes, 2) maintain the hardware context partitioning using runtime feedback from the clients, and 3) service requests from SCAF clients for their current hardware context allocation.

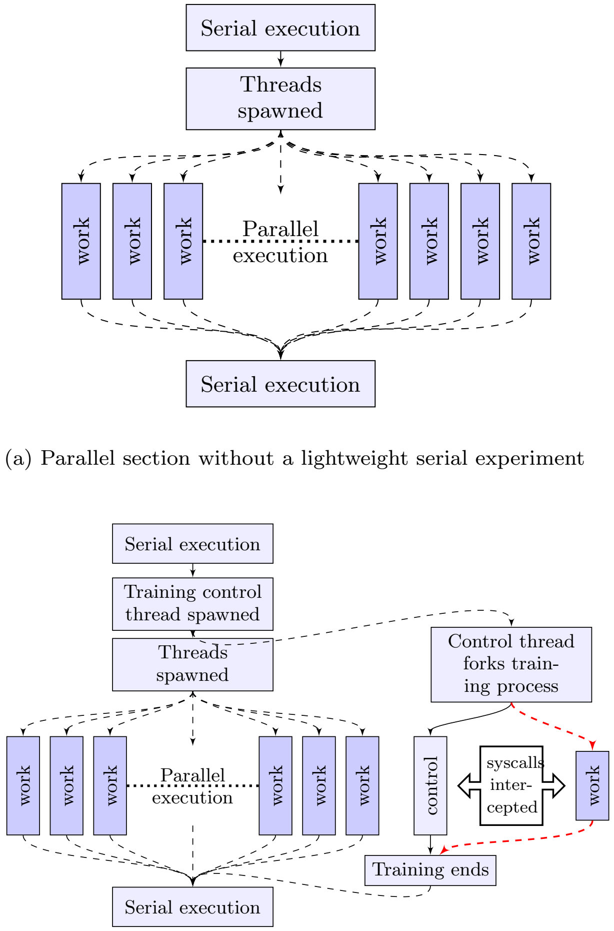
Uncontrollable (non-malleable) process load is monitored through the operating system's facilities. For example, on FreeBSD and Linux, `scafd` monitors the number of kernel timer interrupt intervals (*i.e.*, "ticks" or "jiffies") which have been used by processes which it does not know to be SCAF-compliant processes and uses this to compute the number of hardware contexts which are effectively occupied. This is the same general, inexpensive method used by programs like `top`.

The partitioning of hardware contexts to processes is performed only periodically at a tunable rate, completely asynchronously from clients' requests. The partitioning is performed as described in section 3.3.

Requests from SCAF clients, received over the software bus, arrive in the form of a message containing the client's most recent efficiency metric. This information is stored immediately but not acted upon immediately, since it arrives at a high rate. In order to respond to the requests at the same rate, the daemon periodically evaluates the stored set of client efficiencies and computes a new set of hardware context allocations. This scheme allows the daemon to respond immediately to any requests by returning the latest computed allocation, which may not have incorporated the very latest reported client measurements yet. Other than the initial message a client sends to announce itself to `scafd`, this is the only kind of communication necessary between the clients and the daemon. The rate at which the daemon computes new allocations is tunable, and defaults to 4 Hz, while the rate at which clients check in is variable but generally much higher than 4 Hz.

4.3 The `libgomp` SCAF Client Runtime

The meat of the SCAF implementation lies in the `libscaf`



(a) Parallel section without a lightweight serial experiment

(b) Parallel section with a lightweight serial experiment

Figure 2: Illustration of lightweight serial experiments

client library, but for the sake of clarity it is described here in the context of the `libgomp` client runtime which uses it. The clients perform three interesting functions: 1) recording baseline serial IPC using lightweight serial experiments, 2) recording parallel IPC, and 3) computing parallel efficiency relative to the experiment results as the program runs.

4.3.1 Lightweight Serial Experiments

In SCAF, a lightweight serial experiment allows the client to estimate the serial performance of a parallel section of code. This allows the client to then compute its recent efficiency, and provide a meaningful metric to the SCAF daemon. By default, the client will perform an experiment *only the first time it executes each parallel section, thus reducing its overhead*, although the user is able to tune the client so that it re-runs the experiment periodically. Experiments proceed as follows: given an allocation of N hardware contexts to run parallel section A on for the first time, the `libscaf` will recognize that it has no serial experiment result for A and is due for an experiment run. Provided a function pointer to the parallel section, `libscaf` forks a new *process* which will run A serially on a single hardware context concurrently with the original parallel process. Although the experimental process is a separate proper process, it must share the allocation of N hardware contexts. To accomplish this, `libscaf` simply reduces the number of hardware contexts on which the non-experimental process runs on to $N-1$. The end result is an experimental process running on 1 thread for the sake of measuring its achieved IPC, while the original program still makes progress as usual with $N-1$ threads. Note that the serial execution of the section is not timed, since it may be interrupted early. Instead, its IPC is recorded, since this will still be meaningful. Figure 2 illustrates the lightweight serial experiment technique.

Experiment duration Assuming some speedup is being obtained, the serial experiment process would take longer to complete than the parallel process doing the same work. We cannot afford to wait that long. Thus, we end the experimental process as soon as the parallel process finishes the section. The achieved IPC of the serialized section is recorded in order to compare it to parallel IPC measurements.

Maintaining correctness Since there will be two instances of the original section in execution, care must be taken to avoid changing the machine's state as perceived by its users. The forked experimental process begins as a clone of the original parallel process just before the section of interest. The new process's memory is a copy of the original process's, so there is no fear of incorrectly affecting the original process through memory operations. The only other means a process has to affect system state is through the kernel, by way of system calls. Fortunately, `ptrace(2)` on platforms such as FreeBSD and Linux provides a mechanism for intercepting and denying system calls selectively. On Solaris, the `proc(4)` filesystem can be used to the same effect. Therefore, the experimental process runs until an unsafe system call is requested. For example, a read from a file descriptor is allowed. A series of writes may be allowed, but only if the write is redirected to `/dev/null`. (Nowhere.) A series of writes followed by a read is not allowed, as the read may be dependent on the previous writes, which did not actually occur. Fortunately, parallel sections tend to contain few system calls, and terminating experiments due to unsafe system calls is the exception rather than the norm. For example, none of the NAS NPB benchmarks contain such unsafe system calls in their parallel sections.

Performance of `fork(2)` On modern UNIX or UNIX-like OSs, `fork` only copies the page table entries, which point to copy-on-write pages. This avoids the penalty associated with allocating and initializing a full copy of the parent's memory space. As a result, `fork` is still more expensive than thread initialization, but is not prohibitively expensive when used for serial experiments.

4.3.2 Computing Efficiency

The SCAF runtime calculates an effective efficiency in order to report it back to the SCAF daemon before each parallel section. The client receives an allocation of N threads, which it uses in order to compute the next parallel section. This allocation is considered fixed across any serial execution that occurs between parallel sections. In the OpenMP port, the client constantly collects five items in order to compute its reported efficiency:

1. T_{parallel} : wall time spent inside the last parallel section

2. P_{parallel} : the per-thread IPC recorded in the last parallel section

3. T_{serial} : wall time spent after the last parallel section executing serial code

4. S : an identifier for the last parallel section, generally its location in the binary

5. N : the thread allocation used since the start of the last parallel section

Here it is important to note that T_{serial} **and** T_{parallel} *refer to time spent in different work; in particular, non-parallelized OpenMP code and explicitly parallelized OpenMP code, respectively*, and not to time spent performing the same work. That is, T_{serial} is not related to any lightweight serial experiment measurements.

The client then can compute the following efficiencies, given that it has the serial IPC $P(S)$ of S from a completed lightweight experiment:

$$E_{\text{parallel}} \leftarrow P_{\text{parallel}}/P(S) = \frac{P_{\text{parallel}}N/P(S)}{N} \approx \frac{\text{speedup}}{N}$$

$$E_{\text{serial}} \leftarrow 1/N \approx \frac{\text{speedup}}{N}$$

Since processes report efficiencies only at the beginning of each parallel section, thus $T_{\text{parallel}} + T_{\text{serial}}$ is the time since the last efficiency report to the SCAF daemon. Efficiency since the last report is then estimated as

$$E_{\text{recent}} \leftarrow \frac{E_{\text{serial}} \cdot T_{\text{serial}} + E_{\text{parallel}} \cdot T_{\text{parallel}}}{T_{\text{serial}} + T_{\text{parallel}}}$$

Finally, before being reported to the SCAF daemon, this efficiency value is passed through a simulated RC low-pass filter, with adjustable time constant RC:

$$E \leftarrow \alpha \cdot E_{\text{recent}} + (1-\alpha) \cdot E, \text{ with}$$

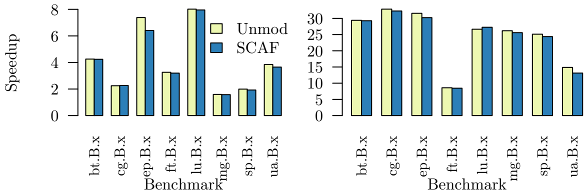
$$\alpha \leftarrow (T_{\text{serial}} + T_{\text{parallel}})/(RC + T_{\text{serial}} + T_{\text{parallel}}).$$

This is a simple causal filter which requires only one previous value to be held in memory. This keeps the efficiency rating somewhat smooth, but at the same time does not punish a process for performing poorly in the distant past. The hope is that the recent behavior of the program will be a good predictor for its behavior in the near future.

5. EVALUATION

In this section, we present results from experiments designed to show the effectiveness of the SCAF system. First, results are provided intended to demonstrate that the overheads imposed by the SCAF runtime and daemon are negligible. Next, we provide a series of results which demonstrate SCAF's ability to improve the efficiency of a multiprogrammed system.

SCAF has been evaluated on two machines representing distinct platforms. The first, "openwks01," is a dual-socket Intel Xeon E5410 system with 8GB main memory running RedHat Enterprise Linux 6 and GCC 4.4.7. The second system, "t2," is a single-socket Sun UltraSparc T2 system with 32GB main memory running Solaris 10 and GCC 4.6.3. The Xeon system, with only 8 cores total, represents a typical small SMP server. The Sparc system, with 64 hardware contexts, represents a more advanced scenario in that many more allocation outcomes are possible.



(a) on a dual Intel Xeon E5410 with 8 hardware contexts

(b) on an UltraSparc T2 with 64 hardware contexts

Figure 3: Speedup of SCAF vs unmodified GOMP

5.1 SCAF-aware GNU OpenMP Port Performance

Since SCAF runtimes work by strictly adding logic at runtime, we provide non-multiprogrammed speedups of the NAS NPB benchmarks with and without SCAF to demonstrate that this overhead is not significant. Benchmarks IS and DC are omitted because they are not malleable without modification. (SCAF detects this and does not attempt to control them.) The remaining 8 benchmarks are tested. *The nature and size of these benchmarks is largely irrelevant as we do not compile or modify them; they are treated as black boxes.* Figure 3 shows that SCAF is generally nearly as fast as the unmodified implementation, with no noticeable slowdown. In general SCAF will be slightly slower than an unmodified runtime due to experiments, which occasionally consume $\frac{1}{N}$ of the system's threads. Note, however, that as hardware grows more parallel, $\frac{1}{N}$ becomes smaller and SCAF's overhead becomes lower. Furthermore, this experimentation is done dynamically only once per parallel section, resulting in very low overhead on programs which run parallel sections repeatedly.

The worst-case scenario for SCAF overhead is realized in the EP benchmark on a small SMP, as seen in Figure 3a. The EP benchmark consists primarily of a single large fork-join section which executes exactly once. Since it is difficult to detect this in a purely run-time system, SCAF executes this section on 7 of 8 threads. The other thread performs a serial experiment in anticipation of future executions of the section, which do not exist in EP. As a result, SCAF incurs a slowdown of about $\frac{7}{8} = 0.875$. We do not include EP in the remaining experiments in this paper since although it is malleable, it cannot be profitably exploited by our current implementation of SCAF. A slightly more advanced implementation will support benchmarks such as EP by modifying the OpenMP scheduler to occasionally consult with SCAF while distributing iterations dynamically. However, this is unimplemented for now, leaving 7 benchmarks amenable to SCAF out of 10 total benchmarks in the NAS suite.

5.2 Improvements in System Efficiency

In this section, we offer results which demonstrate the advantages of deploying SCAF on a multiprogrammed system. We compare three configurations: 1) the unmodified GNU OpenMP implementation, 2) simple equipartitioning, and 3) the fully dynamic SCAF implementation, as described in this paper. In practice, the unmodified configuration is by far the most common since it requires no setup and is readily available. The second configuration, equipartitioning, represents the state of the art which does not require a priori testing. SCAF is the configuration presented in this paper, which also needs no a priori profiling.

5.2.1 Multiprogramming with NPB Benchmark Pairs

For each platform and configuration we evaluated all 21 pairs of 7 NAS benchmarks. In each multiprogramming pair, program 1 first begins execution, and then after 10 seconds program 2 begins execution. This series of events could easily occur when two users are interactively using a remote machine, launching processes while unaware of one another. The 10 second delay was introduced in order to avoid unrealistic precisely-simultaneous starts, but our results are not sensitive to this delay. Problem sizes for each benchmark were chosen such that solo runtimes would be as similar as possible. The average equipartitioned experiment length was 271.6 seconds, with 73% of that time spent multiprogramming.

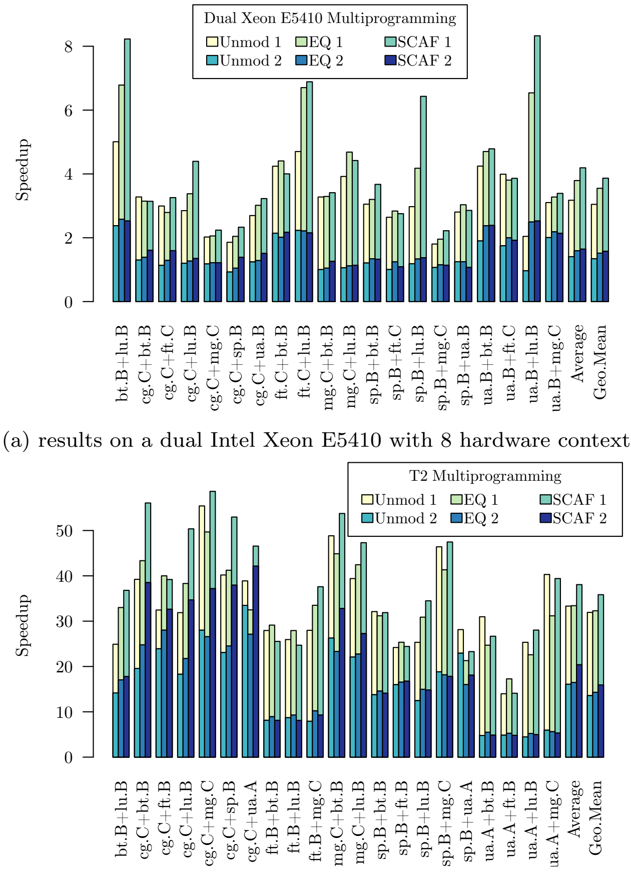
Figure 4a shows all pairwise results on the Xeon-based "openwks01." For 7 benchmarks, 21 pairs exist. The pair (ft.C,mg.C) is missing, as openwks01 did not have enough main memory to run both simultaneously. Here, SCAF shows an improvement over the fastest of unmodified and equipartitioning configurations in 14 out of 20 pairs (70%), with an average improvement of 15% in those cases. The maximum improvement seen is in the pair (sp.B,lu.B), where SCAF yields an improvement of about 53% over equipartitioning.

Figure 4b shows all 21 pairwise results on the UltraSparc T2-based "t2." In this scenario, we see that SCAF offers an improvement in 12 out of 21 pairs (57%), with an average improvement of 15% in those cases. In 3 cases, namely (cg.C,bt.B), (cg.C,lu.B), and (cg.C,sp.B), SCAF offers an improvement of about 30% over the fastest equipartitioning or unmodified result.

We note that in some cases, such as (ft.C,bt.B) in Figure 4a, using IPC as an approximation of work can mislead SCAF to make non-optimal choices. Other times, such as some cases involving ft.B in Figure 4b, SCAF's simple strictly-increasing speedup model leads it to non-optimal decisions: on a T2, ft.B actually slows down beyond 36 cores. However, the performance implications in these cases are small. The fact that not all benchmark pairs benefit from SCAF should not be considered an indictment against it – indeed there is a long history of research into techniques that benefit only a class of applications[2]. With this in mind, benefiting 50-70% of application pairs significantly is quite good.

5.2.2 Detailed 3-Way Multiprogramming Scenario

The previous section (5.2.1) focuses on 2-way multiprogramming, but SCAF can handle greater numbers of clients. The severity of oversubscription seen in the unmodified configuration only increases with increased multiprogramming. In this section, we focus on a particular 3-way multiprogramming experiment on the UltraSparc T2. When run by itself, the first benchmark, cg.C, scales extremely well on a T2, with a maximum speedup near 50X on 64 threads. The other two benchmarks, sp.B and lu.B, scale more modestly, with maximum speedups of 25-30X on 64 threads.



(a) results on a dual Intel Xeon E5410 with 8 hardware contexts

(b) results on an UltraSparc T2 with 64 hardware contexts

Figure 4: 2-way multiprogramming results with 7 NAS NPB benchmarks

In this experiment, cg.C begins at time 0, then sp.B 10 seconds later, and lu.B after an additional 10 seconds. We executed this workload for each of the 3 configurations. Table 5a summarizes the results, while Figure 5b plots log output of the SCAF daemon (`scafd`) after running this scenario.

In Table 5a, we see that the unmodified configuration manages a mediocre sum speedup of 31.5X due to severe oversubscription. Each benchmark uses 64 threads. As the third benchmark begins, the SunOS scheduler is left to timeshare 192 CPU-bound threads to 64 hardware contexts. This severe oversubscription results in lackluster performance from all three benchmarks due to context switching and hardware contention.

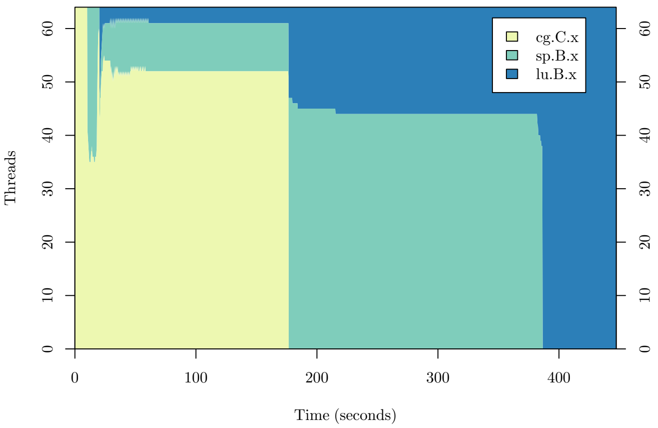
The equipartitioning configuration manages to avoid oversubscription and as a result completes all three benchmarks faster. At first, cg.C runs on all 64 threads. After 10 seconds, sp.B begins and is given 32 of cg.C's threads. Finally, after lu.B begins, sp.B and lu.B are allocated 21 threads while cg.C is allocated 22 threads. lu.B finishes first, leaving cg.C and sp.B each 32 threads. Finally, cg.C finishes, leaving sp.B to complete with all 64 threads. Here, the sum speedup increases to 40.7X due to a lack of oversubscription.

With SCAF, we see improved overall performance. Figure 5b shows SCAF's allocations throughout the experiment. Initially, cg.C runs on all 64 threads. After 10 seconds, sp.B begins and is briefly allocated approximately 29 of the threads. At 20 seconds, sp.B begins as well. Very quickly, SCAF's begins to observe that cg.C is scaling particularly

[2] e.g., faster garbage collectors only benefit benchmarks with heap data, and among those, only those with significant garbage – however garbage collection is still worthwhile.

Configuration	Process	Runtime	Speedup	$\sum$ Speedup
Unmodified	CG	435.9s	13.1	31.5
	SP	474.6s	9.6	
	LU	507.3s	8.8	
Equi-partitioning	CG	374.0s	15.5	40.7
	SP	380.8s	12.2	
	LU	349.8s	13.0	
SCAF	CG	172.2s	35.7	59.3
	SP	374.0s	12.5	
	LU	424.0s	11.1	

(a) Summary of the 3-way multiprogramming scenario



(b) SCAF behavior with 3-way multiprogramming

Figure 5: 3-way multiprogramming example

well, resulting in a significant allocation of 52 threads. Due to this large allocation, cg.C finishes after only 172.2 seconds, after which sp.B and lu.B are allowed to expand to use the full 64 contexts. At this point, sp.B and lu.B are observed to have similar performance, with sp.B having shown better results while cg.C was running. As a result, SCAF allocates about 44 threads to sp.B and the remaining 20 go to lu.B. Finally, at 384 seconds sp.B finishes and lu.B is left to complete on all 64 threads. The achieved sum speedup is 59.3X, an 88% and 45% improvement over the unmodified and equipartitioning configurations, respectively.

6. FUTURE WORK

We are investigating extending this work in three ways:

Porting additional runtime systems.

We intend to implement SCAF for additional runtime systems beyond GNU OpenMP, such as Open64's OpenMP runtimes and Intel's TBB library. Although most of the runtime changes will be very similar, some differences will arise for TBB. TBB makes use of a dynamic work-stealing model, which may result in design changes when modifying TBB to support changing the number of threads used at runtime and require additional methods for estimating parallel efficiency. However, OpenMP is the de-facto standard for shared memory programs today, and is far more prevalent.

Expanding results to additional hardware platforms.

SCAF has been primarily tested on Intel x86_64 SMPs. We are investigating several more platforms, including Tilera's mesh-connected grid processors, SGI UV 2000 NUMA systems with 512-1024 cores, and Intel Xeon Phi 5110P coprocessors with 60 cores and 240 threads. Preliminary ports of SCAF to these platforms have been created and are promising. SCAF and its techniques are intended to become more useful and effective on more parallel systems such as these.

Periodic lightweight experiments.

One advantage of having a method for collecting serial IPC at runtime is that it allows the measurement to be repeated periodically or on certain triggers. Some long-running processes may have serial IPC which is very dependent upon input. In these cases where inputs can vary greatly, it may not even be possible to gather comprehensive information on serial performance even we are able to run tests ahead of time. A SCAF system which can re-run serial experiments would be able to overcome these difficulties.

7. CONCLUSION

This work has shown that neither a priori testing, nor simple equipartitioning is generally satisfactory. We argue that none of the related work has caught on in practice, due to the significant inconvenience to the user of performing profiling or testing ahead of time, or because they require changes to the program or re-compilation. We have presented a drop-in system, SCAF, which includes a technique for collecting equivalent information at runtime, paying only a modest performance fee and enabling sophisticated resource management without recompilation, modification, or profiling of programs. We believe that such resource management will be important as hardware becomes increasingly parallel, and as more parallel applications become available.

8. ACKNOWLEDGMENTS

This work was supported by a NASA Office of the Chief Technologist's Space Technology Research Fellowship. We would also like to thank UMIACS for providing access to the Xeon-based "openwks01," part of the UMIACS OpenLab.

9. REFERENCES

[1] T. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 6 –16, Jan. 1990.

[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '98. New York, NY, USA: ACM, 1998, pp. 119–129. [Online]. Available: http://doi.acm.org/10.1145/277651.277678

[3] R. D. Blumofe and D. Papadopoulos, "Hood: A user-level threads library for multiprogrammed multiprocessors," Tech. Rep., 1998.

[4] 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,” *SIGMETRICS Perform. Eval. Rev.*, vol. 22, no. 1, pp. 33–44, May 1994. [Online]. Available: http://doi.acm.org/10.1145/183019.183023

[5] A. B. Downey, *A model for speedup of parallel programs.* University of California, Berkeley, Computer Science Division, 1997. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1997/CSD-97-933.pdf

[6] M. Frigo, C. E. Leiserson, and K. H. Randall, “The implementation of the cilk-5 multithreaded language,” *SIGPLAN Not.*, vol. 33, no. 5, pp. 212–223, May 1998. [Online]. Available: http://doi.acm.org/10.1145/277652.277725

[7] M. W. Hall and M. Martonosi, “Adaptive parallelism in compiler-parallelized code,” *Concurrency: Practice and Experience*, vol. 10, no. 14, pp. 1235–1250, Dec. 1998. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/(SICI)1096-9128(19981210)10:14<1235::AID-CPE373>3.0.CO;2-Z/abstract

[8] J. Hungershöfer, A. Streit, and J.-m. Wierum, *Efficient Resource Management for Malleable Applications*, 2001.

[9] ICL, University of Tennessee and contributors, “Performance application programming interface (PAPI),” 2012. [Online]. Available: http://icl.cs.utk.edu/papi/

[10] iMatix Corporation and contributors, “ZeroMQ,” 2012. [Online]. Available: http://zero.mq/

[11] L. V. Kale, S. Kumar, and J. DeSouza, “A Malleable-Job system for timeshared parallel machines,” in *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2002.* IEEE, May 2002, pp. 230– 230.

[12] I. H. Kazi and D. J. Lilja, “A comprehensive dynamic processor allocation scheme for multiprogrammed multiprocessor systems,” in *2000 International Conference on Parallel Processing, 2000. Proceedings.* IEEE, 2000, pp. 153–161.

[13] D. J. McFarland, *Exploiting Malleable Parallelism on Multicore Systems.* [Blacksburg, Va: University Libraries, Virginia Polytechnic Institute and State University, 2011. [Online]. Available: http://scholar.lib.vt.edu/theses/available/etd-06292011-130247

[14] H. Pan, B. Hindman, and K. Asanović, “Composing parallel software efficiently with lithe,” in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI ’10. New York, NY, USA: ACM, 2010, pp. 376–387. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806639

[15] J. H. Schonherr, J. Richling, and H.-U. Heiss, “Dynamic teams in OpenMP.” IEEE, Oct. 2010, pp. 231–237. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5644944

[16] R. Sudarsan and C. Ribbens, “ReSHAPE: a framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment,” in *Parallel Processing, 2007. ICPP 2007. International Conference on*, Sep. 2007, p. 44.

[17] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, “Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps,” in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 277–286. [Online]. Available: http://doi.acm.org/10.1145/1346281.1346317

[18] A. Tucker and A. Gupta, “Process control and scheduling issues for multiprogrammed shared-memory multiprocessors,” in *Proceedings of the twelfth ACM symposium on Operating systems principles*, ser. SOSP ’89. New York, NY, USA: ACM, 1989, pp. 159–166. [Online]. Available: http://doi.acm.org/10.1145/74850.74866