Efficient Multiprogramming for Multicores with SCAF

Abstract
As hardware becomes increasingly parallel and the availability of scalable parallel software improves, the problem of managing multiple multi-threaded 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 runtime 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 allocation, or does not account for process efficiency at all. This paper presents the Scheduling and Allocation with Feedback (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, and demonstrate that it produces consistent results. We show that the overhead of SCAF is negligible.

In one measured multi-process scenario, the system throughput (as measured by sum of speedups) improved by nearly 3X using SCAF compared to using unmodified OpenMP libraries. Furthermore, in the same scenario, SCAF improved throughput by 26% versus equipartitioning. Finally, SCAF is able to equal the throughput of a system that uses a priori profiling, but without any profiling or other user involvement.

Categories and Subject Descriptors D.4.1 [Operating Systems]: Process Management—Multiprocessing, Scheduling, Threads; D.3.4 [Programming Languages]: Processors—Run-time environments

General Terms Algorithms, Design, Languages, Performance

Keywords Parallelism, multi-threaded multiprogramming, resource management, user-Level scheduling, oversubscription

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[4, 10], while others do not require profiling and do not account for program efficiency[5, 11, 13, 15]. 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 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, and running both simultaneously for the duration of the current parallel section. 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; 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.

We demonstrate that the SCAF runtime incurs low overhead on a process running alone, despite experiments. We also provide results showing that the lightweight experiments produce consistent and useful results. Furthermore, we evaluate the SCAF system in a multiprogramming scenario, showing that SCAF improves system efficiency over simple equipartitioning, and approaches the system efficiency obtained when profiling information is already available.

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 multi-threaded 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 [9] 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 [14] improved on this work with ReSHAPE, their framework for dynamic resizing and scheduling. The ReSHAPE system assumes that the parallel application is iterative. Using the provided resizing library and its associated 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 in order to use their API.

2.2 Shared Memory Systems

Relatively little work has been done concerning multiprogrammed, multi-threaded scheduling and the problem of oversubscription. Tucker et al [15] 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 (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.

Hall et al [4] 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 [10] 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 dur-
Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicates to the runtime system via API calls between parallel sections. Once recompiled, the application communicating program execution, avoiding the need for careful application profiling.

Iancu et al [6] takes a different approach, performing experiments with allowing hardware oversubscription and relying only upon the operating system’s ability to timeshare hardware contexts. They found that when using this technique with simultaneous parallel workloads, oversubscription can in many cases outperform fair hardware partitioning, and they advocate oversubscription as a solution for multiprogrammed multi-threaded environments. They note that oversubscription performance depends heavily on the collective and barrier implementations used. SCAF implements a more generic solution which can tolerate collective and barrier implementations which depend on having a dedicated hardware context per thread for good performance. Additionally, we argue that while depending on time-sharing in the face of oversubscription may be tolerable on smaller multiprocessors, it is not a good solution for many-core systems. Also, we observe that when oversubscription helps, it helps a little; but when it hurts, it hurts a lot. As a result, SCAF strictly avoids oversubscription.

More recently, Pan et al [12] 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 N² 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 [11] 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 IPC¹ 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 [13] 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 [5] 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.

6. **Lithe**, [12]
7. **A comprehensive dynamic processor allocation scheme for multiprogrammed multiprocessor systems**, [10]

---

¹ Instructions per cycle
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. They load SCAF-compliant runtime libraries in place of the stock 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. The SCAF daemon is responsible for accounting for the activities of the non-malleable processes.

3.2 Conversion from Time-sharing to Space-sharing

By default, modern multi-user operating systems support the execution of multiple multi-threaded 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, with 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” to be \( E \equiv \frac{S_j}{p_j} \).

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.

Equi-partitioning

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:

\[
p_j \leftarrow \left\lfloor \frac{N}{k} \right\rfloor
\]

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 \( S_A(p_A) = 1 + \frac{1}{3}(p_A - 1) \) and program \( B \) with \( S_B(p_B) = 1 + \frac{1}{3}(p_B - 1) \), 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.

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 \( \sum_j S_j(p_j) \) 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 [7] runtime library, since this is generally available from hardware counters. 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. However, we chose to use the more generic metric of IPC in order to avoid limiting SCAF’s usefulness to 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

\[ S_j(p_j) \approx 1 + C_j \log p_j, \text{ where } C_j \leftarrow \frac{E_j p_j - 1}{\log p_j} \]

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 speedup curve specified by tuples (number of threads, 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 [2, 3]. 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 constant \( C_j \) can be adjusted to approximate speedup curves of real applications using only a single measurement representing recent efficiency. A dynamic system which fits to multiple distinct speedup points into account 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

\[ \max_p \left\{ \sum_{j=1}^{k} S_j(p_j) \left| p > 0 \land \sum_{i=1}^{k} p_i = N \right. \right\} \]

or, equivalently using equation 2,

\[ \max_p \left\{ \sum_{j=1}^{k} 1 + C_j \log p_j \left| p > 0 \land \sum_{i=1}^{k} p_i = N \right. \right\} \]

By breaking the constant 1 out of the summation and taking advantage of the fact that \( \log a + \log b = \log ab \), we can express the optimization problem as

\[ \max_p \left\{ k + \log \left( \prod_{j=1}^{k} p_j^{C_j} \right) \left| p > 0 \land \sum_{i=1}^{k} p_i = N \right. \right\} \]

Since in general \( \log(x) \) is strictly increasing with \( x \) for \( x > 0 \), we can further reduce the problem to

\[ \max_p \left\{ \prod_{j=1}^{k} p_j^{C_j} \left| p > 0 \land \sum_{i=1}^{k} p_i = N \right. \right\} \]

It can be shown that this problem has the closed-form solution

\[ p_i = \frac{NC_i}{\sum_{j=1}^{k} C_i} \]

For the sake of preserving space, the full proof of equation 7 is not shown here. However, giving the objective function the name \( f \), it proceeds as follows:

1. eliminate \( p_i \) by expressing it as \( N = \sum_{i=1}^{k} p_i \) in \( f \)
2. compute the set of first-order partial derivatives \( \frac{\delta f}{\delta p_1}, \ldots, \frac{\delta f}{\delta p_{k-1}} \)
3. show that the first-order derivatives all take value 0 only at the proposed solution point, resulting in a lone critical point
4. show that \( f \) is a strictly concave function by proving its Hessian is everywhere negative definite
5. since \( f \) is strictly concave, the critical point must be a global maximum of \( f \)

We have also verified the solution in equation 7 offline, symbolically for various values of \( k \) with Mathematica, and numerically for many values of \( k \) and \( C \) with Maple. SCAF itself uses equation 7; it does not use these software packages.

As per equation 7, the SCAF daemon sets the allocations \( p \) by computing the vector \( C \), then assigning \( p_i \) to be the fraction \( C_i / \sum_j C_j \) 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 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. 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 \( C_{foo} = 0.19\) and \( C_{baz} = 0.32\) and compute new allocations \( p_{foo} = 0.14 \) and \( p_{baz} = 0.18 \). 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.
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-SCAF) 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 *libgomp* 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 *libgomp*, but for the sake of clarity it is described here in the context of *libgomp*, 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, 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 *libgomp* 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, *libgomp* 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 $N$ hardware context allocation. To accomplish this, *libgomp* 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.

---

**Figure 1: Runtime feedback loop in SCAF’s partitioning scheme**

For example, if $foo$ scales better than the $1 + \frac{N-1}{\log 14} \log 14 = 8.43X$ 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. These minor changes call into a *libscaf* client library which is designed to easily support development of additional runtime ports with similar ease. It is expected that other OpenMP runtimes (such as Open64’s) can be supported with similar ease. 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 [8]. 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
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.

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. 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 paralleled 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:

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

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

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

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

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

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

with

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

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. Second, we demonstrate that our lightweight serial experiment method provides consistent and accurate results and is effective at collecting serial performance information in the face of other system activity. Finally, we provide a series of results which demonstrate SCAF’s ability to improve the efficiency of a multiprogrammed system.

The system used for the experiments was a dual-socket AMD Opteron 6212 (16 cores total) with 128GB main memory, running a 64-bit Linux 3.2.0 kernel. Binaries were compiled by GCC 4.6. A single set of binaries were used for all tests, as SCAF does not require modification or recompilation.

5.1 SCAF-aware GNU OpenMP Port Performance
Since SCAF runtimes work by strictly adding logic at runtime, we ran a series of tests from the NAS NPB benchmark suite in order to demonstrate that this overhead is not significant. Benchmarks FT and MG are omitted due to runtime errors when using any nontrivial problem sizes, such as “C” or larger. (This problem was observed on several platforms, with or without SCAF.) Benchmarks IS and DC are omitted because they are not malleable without modification. The remaining 6 benchmarks are tested. Figure 3 shows that SCAF is generally nearly as fast as the stock implementation, with no noticeable slowdown. In general SCAF will be slightly slower than a stock 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 only once per parallel section, resulting in very low overhead on programs which run parallel sections repeatedly.

5.2 Lightweight Serial Experiment Precision
To demonstrate that the lightweight experiment mechanism can accurately estimate serial performance of a section of parallel code no matter the program’s current allocation, we recorded measurements obtained by the SCAF-aware \$\texttt{1\_}\texttt{b\_gomp}\$ port for each of the eight parallel sections (A-H) in the NAS “CG” benchmark. All measurements were taken by calls to the PAPI library. Figure 4 shows the results of this experiment. Each value along the x-axis represents a full execution of CG on some number of threads, and the corresponding IPC values along the y-axis are the results of lightweight experiments for each of the 8 parallel sections in CG. Since measurements should ideally not vary with the program’s allocation, the lines corresponding to parallel sections should be as flat as possible.

Figure 3: Speedup of SCAF vs stock GOMP

![Figure 3](image)

We see in Figure 4 that sections can vary in IPC, and that the measurements reported by lightweight experiments do not vary greatly as the program varies its allocation. Similarly promising results (not shown) were seen when running the rest of the NAS benchmarks. These accurate measurements allow the client to make good estimates of its own efficiency, entirely at runtime.

5.3 Improvements in System Efficiency
In this section, we offer results which demonstrate the advantages of deploying SCAF on a multiprogrammed system. We compare four configurations: A) the stock GNU OpenMP implementation, B) simple equipartitioning, C) SCAF with experiment results substituted for a priori measurements, and D) the fully dynamic SCAF implementation, as described in this paper. In practice, configuration A is by far the most common since it requires no setup and is readily available. The second configuration, B, represents the state of the art which does not require a priori testing. Configuration C represents the state of the art if we assume that a priori profiling is acceptable. Finally, configuration D is the configuration presented in this paper, which needs no a priori profiling.

5.3.1 Multi-process Scenario
This scenario simulates two users each launching a parallel process at the same time. In this scenario, the users are operating remotely and are unaware of one another’s intentions, so coordination is not possible. The first user launches the NAS “UA” benchmark, while the other launches the NAS “BT” benchmark. However, most NAS benchmarks, including UA and BT, have OpenMP implementations that achieve good speedups, yielding uninteresting allocations. In order to demonstrate system behavior when one parallel process is quite inefficient, we compiled the serial version of BT with the AESOP auto-parallelizing compiler, using OpenMP as the parallelization technique. AESOP parallelized 17 sections of code. However, in this particular case, AESOP’s lack of advanced transformations such as array privatization prevent it from parallelizing a few key sections, resulting in an inefficient parallel binary.

Figure 5 compares the behavior of the four configurations in this scenario, while Table 2 summarizes the results. Table 2 shows that SCAF (D) equals the overall sum-of-speedups of a system requiring a-priori profiling (C), without such profiling. SCAF also has about 26% greater sum-of-speedups than equipartitioning (B), and nearly 3X greater sum-of-speedups than stock OpenMP (A). Note that nearly all multicore today have only configuration A available.

Figure 5, depicting the most common configuration (A), shows immediate severe oversubscription, resulting in very poor performance due to context switching and hardware contention. Both pro-
Figure 5: Behavior of SCAF with a complex workload

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Process</th>
<th>Runtime</th>
<th>Speedup</th>
<th>∑ Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: Stock</td>
<td>UA</td>
<td>666.2s</td>
<td>1.80</td>
<td>2.28</td>
</tr>
<tr>
<td></td>
<td>BT</td>
<td>840.5s</td>
<td>0.48</td>
<td></td>
</tr>
<tr>
<td>B: Equipartitioning</td>
<td>UA</td>
<td>279.6s</td>
<td>4.29</td>
<td>5.31</td>
</tr>
<tr>
<td></td>
<td>BT</td>
<td>401.3s</td>
<td>1.02</td>
<td></td>
</tr>
<tr>
<td>C: A priori profiling</td>
<td>UA</td>
<td>208.5s</td>
<td>5.76</td>
<td>6.77</td>
</tr>
<tr>
<td></td>
<td>BT</td>
<td>406.8s</td>
<td>1.01</td>
<td></td>
</tr>
<tr>
<td>D: SCAF</td>
<td>UA</td>
<td>210.0s</td>
<td>5.71</td>
<td>6.72</td>
</tr>
<tr>
<td></td>
<td>BT</td>
<td>406.2s</td>
<td>1.01</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Summary of results in the multi-process scenario

Processes run 16 threads all the time for a total of 32 threads, whereas the machine only has 16 hardware contexts.

Configuration B, where simple equipartitioning is used, manages to avoid oversubscription and as a result completes both tasks faster. Here Figure 5b shows that initially both threads share the 16 hardware contexts equally with 8 threads each. When BT terminates, UA uses all 16 contexts. The drawback is that BT receives the same share of resources as UA despite its occasionally lower efficiency.

Configuration C, which has the advantage of a priori profiling, demonstrates improved overall performance. Figure 5c shows that initially BT is allocated only 4 threads because of its lower efficiency whereas the more-efficient UA gets 12 threads. When UA terminates, BT gets 16 threads. Overall system efficiency improves, yielding a higher sum-of-speedups in Table 2. However, this configuration has the significant disadvantage of requiring the end users to manually run profiling ahead of time. This is unrealistic to expect from most users.

Finally, in Figure 5d, configuration D (SCAF) shows that its behavior does not deviate significantly from that of configuration C. Since a small amount of resources are spent on measuring serial IPC, SCAF is not expected to surpass configuration C in any aspect. However, we see that SCAF is able to approach the behavior of configuration C while not requiring the system’s end users to manually run profiling on their programs.

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 large Intel x86_64 SMPs. We have recently acquired access to several interesting Tilera-based processors, which make use of square grids of 36–64 cores each. SCAF and its techniques are intended to become more useful and effec-
tive on more parallel systems such as these. The primary reason that these results are not already included is the lack of a PAPI port for these processors.

**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.

**References**


