Efficiently Combining Parallel Software Using Fine-grained, Language-level, Hierarchical Resource Management Policies

Zachary Anderson

Systems Group, ETH Zürich
Zürich, Switzerland
zachary.anderson@inf.ethz.ch

Abstract

This paper presents Poli-C, a language extension, runtime library, and system daemon enabling fine-grained, language-level, hierarchical resource management policies. Poli-C is suitable for use in applications that compose parallel libraries, frameworks, and programs. In particular, we have added a powerful new statement to C for expressing resource limits and guarantees in such a way that programmers can set resource management policies even when the source code of parallel libraries and frameworks is not available. Poli-C enables application programmers to manage any resource exposed by the underlying OS, for example cores or IO bandwidth. Additionally, we have developed a domain-specific language for defining high-level resource management policies, and a facility for extending the kinds of resources that can be managed with our language extension. Finally, through a number of useful variations, our design offers a high degree of composability. We evaluate Poli-C by way of three case-studies: a scientific application, an image processing webserver, and a pair of parallel database join implementations. We found that using Poli-C yields efficiency gains that require the addition of only a few lines of code to applications.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming

General Terms Languages, Performance

Keywords language extension, resource management, policy, hierarchical parallelism, scheduling

1. Introduction

Due to recent trends in computer architecture, previously single-threaded applications are making increasing use of parallel libraries and frameworks in order to achieve steady performance gains. Setting aside the commonly accepted idea that parallel programming is more difficult than sequential programming due to concurrency concerns, parallel programming is also difficult because machine resources such as cores, caches, disks, and network interfaces may not be managed by the operating system in a way that makes sense for the application. This is especially true when an application uses multiple parallel libraries or frameworks simultaneously, as is the case both with new implementations of traditional scientific applications [14, 31], and with upcoming applications that include "recognition, mining, and synthesis" tasks [8, 10, 40].

In this paper we describe Poli-C, a language extension and runtime that exposes new OS resource management features to application programmers with the goal of allowing applications to tailor resource management to their own needs rather than relying on the one-size-fits-all approach supplied by modern operating systems. In particular, we have added a powerful new statement to C that provides resource guarantees and limits in such a way that programmers can set resource management policies for, and among, hierarchically parallel applications. Additionally, we have developed a domain-specific language for defining high-level resource management policies, and a facility for extending the kinds of resources that can be managed with our language extensions.

In particular, we introduce a new statement to C having the following syntax: `require(r(a)) {...}`. In its most basic form, this statement indicates that inside of the block of code, the program has exclusive access to the amount a of resource `r`, with the limitation that it cannot exceed that amount. If the requested resource is not available, the program releases the resources it already holds and blocks waiting for it. Additionally, using special functions that we call *policies* programs may request not only single resources, but also sets of resources after querying resource availability and contention at runtime.

The design of Poli-C provides three main benefits. First, it offers composability. Using a number of options to the `require` statement, Poli-C can efficiently combine off-the-shelf, unmodified parallel libraries. This paper describes in detail the design and implementation of the `require` statement and the ways its operation is modified by the options it accepts.

Secondly, our design allows fine-grained management of any resource exposed by the operating system, for example cores and IO bandwidth. If the underlying OS provides an API for measuring or managing a resource, then, after writing a few lines of glue code, that resource can be managed through the runtime and language extensions provided by Poli-C. Among threads in a single process, we achieve this by decoupling resource allocation from thread scheduling through the use of explicit *allocation trees*. Among all processes, we achieve this by providing a system-level daemon, `policd`, that tracks resource usage across the entire system. This paper describes the operation of our allocation trees and `policd`.

Exposing this low level of control to the application programmer might be seen as reversing a trend toward increased abstraction of hardware resources by operating systems and virtual machines. This trend likely makes many applications easier to build and maintain. However, applications with demanding performance requirements, such as databases, webservers, and scientific codes, *already* subvert OS and VM provided abstraction boundaries, albeit in an ad-hoc, system-specific way. One important goal achieved by Poli-C is to allow this abstraction boundary to instead be shifted in a principled, portable, and reusable way that is made explicit in the source code of these applications through the `require` statement.

Finally, our design allows us to separate policy from implementation. With our approach, policies are declared using our policy DSL, but implemented as part of the language runtime, transparent to the application programmer, but still available for modification using declarative features that, thanks to Poli-C, are now part of the language. This paper describes our policy DSL, and evaluates its use in a number of examples.

Our work is inspired by the Lithe hierarchical scheduling framework [36]. Since the default OS scheduler is unaware of high-level tasks involving groups of cooperating threads, its thread scheduling is inefficient. Lithe solves this problem by enabling the allocation of cores to high-level tasks, rather than to individual threads. Poli-C solves the same problems as Lithe with three key improvements. First, Poli-C manages other resources in addition to cores. Second, using `policd`, it manages resources not only within one multithreaded process, but also across multiple multithreaded processes. Finally, through the use of options to the `require` statement, it is possible to use Poli-C to compose parallel libraries without modifying and creating custom builds of each parallel library to be composed.

In summary we make the following contributions:

- The design and implementation of a runtime library, language extension, and system daemon that enable fine-grained, language-level, hierarchical resource management policies suitable for use among applications that compose parallel libraries and frameworks.
- The gathering of disjoint OS APIs and interfaces into one coherent tool with a more user-friendly, less error-prone, language-based interface
- The design and implementation of a domain-specific language for defining resource allocation policies.
- The evaluation of Poli-C in three different settings: an important scientific application, an image processing webserver, and a pair of parallel database join implementations. We found that using Poli-C yields efficiency gains requiring the addition of only a few lines of code to applications.

The rest of the paper is organized as follows. We begin by describing many of the features of our language extensions by way of a simple example (Section 2). Then, we discuss some of the high-level design issues that came up in building Poli-C (Section 3). This is followed by a more detailed description of the syntax and semantics of the Poli-C extensions, and the features we have included for managing new resources and defining new policies (Section 4). Next, through a more in-depth example, we describe the features of Poli-C that aid composability. Having described all of Poli-C's features, we give a detailed description of the implementation of our runtime, which includes a description of our allocation trees, and `policd` (Section 6). In Section 7 we demonstrate the usefulness of Poli-C, and show some of its performance advantages. Finally, we explore related work (Section 8), and conclude with a discussion of future directions (Section 9).

2. Overview

This section describes the operation of Poli-C in the context of a simple example. In particular, we have added a statement called `require` to C. The `require` statement is parameterized by a list of *resource kinds* and *policies*, which we describe below. Based on this list, the `require` statement provides both a resource guarantee and a limit to a block of code. Our runtime system enforces the property that all effects in the block of code under the `require` statement receive the resource guarantee and respect the resource limit.

Consider the program in Figure 1. In this example, we will use Poli-C to ensure that only two of a program's threads run at any given time, and that they do so with exclusive access to one core each. This may be a desirable arrangement

```
 1 void *f(void *a) {
 2   cores_t c;
 3   require(c=cores(1)) {
 4     int ncores = c.ncores;
 5     ...
 6   }
 7 }
 8
 9 int main() {
10   require(cpuUtil(0, 1.0), cpuUtil(1, 1.0),
11           cpuUtil(2, 0.0), cpuUtil(3, 0.0)) {
12     for(i = 0; i < n; i++)
13       spawn f(i);
14     join_all();
15   }
16 }
```

Figure 1. Example program using our extensions to C.

if, for example, the application programmer wishes to prevent a thread's cache from being polluted by other threads.

2.1 Resource Kinds

The `main` function of this program spawns off `n` threads that each execute function `f`. Poli-C's extensions are used in this program on lines 3 and 10. Assuming that the code in the listing is running on a machine with four cores, the `require` statement on line 10 indicates that the enclosed code may use up to 1.0, i.e. 100%, of cores 0 and 1, but may not use cores 2 and 3 since the requested utilization for them is 0.0, i.e. 0%. Further, it guarantees that 100% utilization on cores 0 and 1 will be available for the duration of the code block.

Here, `cpuUtil` is a *resource kind*. That is, it represents a particular type of resource, and causes the runtime system to use specialized operations defined for that resource kind. These specialized operations enforce the invariant that a block of code does not consume more resources than it has been allocated. The resource kinds are not baked into Poli-C, and new ones may be provided. In Section 3 we discuss who may define resource kinds under different usage scenarios.

When there is more than one device of a resource kind, the resource kind is parameterized by an identifier for a specific device, in this example, a numeric core ID. The mapping from identifiers to devices is defined when a resource kind is created, which we describe in in Section 4.2. For now it suffices to point out that the resource kinds are initialized when a program starts, after using OS calls to query the available hardware, and that `cpuUtil` is parameterized by core ID and governs how much CPU utilization on a particular core is allocated to a block of code. Making requests directly through resource kinds lacks flexibility. This inflexibility is addressed by our *policies*, described shortly.

2.2 Initial State

Before a program executes a `require` statement, it is constrained to a limited set of resources. These constraints are configurable, and are enforced by our system-level daemon, `policd`, which we describe in detail in Section 6.3. For now however, in order to simplify discussion of the semantics of the `require` statement for this high-level overview, we assume that there is only one process running in the system, and that at its start, `policd` allocates it all available system resources. In Section 6.3 we discuss how this assumption is eliminated so that Poli-C may provide its invariants to all running processes.

Having made this assumption, before a `require` statement is executed, all threads share all resources at the discretion of the OS scheduler. A `require` statement has the effect of carving out some resources from these shared resources for exclusive use in a block of code. That is, if nothing about a resource kind is mentioned in a `require` statement, then no change in sharing takes place for that resource kind. Therefore, inside of the `require` statement on line 10, the code has exclusive access to cores 0 and 1, but if the `require` statement had not mentioned the other cores, then those cores would still have been shared among all threads as mediated by the OS. On the other hand, since the `require` statement makes a request for 0% utilization on cores 2 and 3, code in the `require` statement is prevented from using them by OS calls made by the implementation of the `cpuUtil` resource kind.

2.3 Spawning Threads

Inside of the `require` statement on line 10, the program spawns `n` threads. Since these threads are spawned within the `require` statement, they also have access to the resources specified in the `require` statement. In particular, they share cores 0 and 1 with each other and with the parent thread, but they may not use cores 2 and 3.

When child threads are spawned inside a `require` statement, they share the parent's guaranteed resources, and inherit the parent's limitations. That is, the parent thread, and threads spawned within the same `require` statement share evenly the allocation received through the `require` statement. If a spawned thread subsequently enters its own `require` statement, the allocation it receives is no longer shared with any other thread. That is, we maintain the invariant that allocations granted by the `require` statement are exclusive until shared by spawned threads, and that shared resources remain shared until exclusive access is granted by a `require` statement.

If the parent thread leaves the `require` statement while the threads spawned within it still exist, the child threads retain the guarantees and limitations of the `require` statement. A consequence of this is that the parent thread may not obtain a guarantee through a new `require` statement for the same resources that were promised to child threads spawned in an earlier `require` statement. Otherwise the resource guarantees for the child threads could be violated. When the child threads no longer exist, the parent thread may re-`require` the resources they were using.

For example, if the `join_all()` call on line 14 were moved outside of the `require` statement, the child threads would still have exclusive access to cores 0 and 1, and not would not have access to cores 2 and 3. On the other hand, the parent thread would have access to cores 2 and 3, but not 0 and 1 any longer, so that the guarantee of exclusive access would still be provided to the child threads.

2.4 Policies

At the `require` statement on line 3 each thread makes a request for exclusive access to a single core. Here, `cores` is a *policy*. Since requesting a statically determined amount of each individual resource is neither convenient nor portable, Poli-C offers *policies* as a way to choose a set of resources at runtime based on availability and contention. Policies are defined using a domain-specific language that we describe in Section 4.3. For now it suffices to point out that the `cores` policy examines the unallocated utilization available in all devices of the `cpuUtil` resource kind, and attempts to `require` 100% utilization on some number of them, according to the parameter it is passed. On the other cores, it requests 0% so that they will not be used. The `cores` policy has the effect of requesting that the code in the block have exclusive access to some number of cores.

2.5 Policy Results

Since resource levels requested by policies are determined at runtime, we provide a facility for communicating the results of a policy back to the application. In particular, policies may define a type to be used for this communication. On line 4, the program reads the field `ncores` of the local `c` into a local variable. The type `cores_t` is defined along with the `cores` policy, and a variable of this type may be specified in the `require` statement using the syntax indicated. The variable, having been written by the policy function, is then in scope inside of the `require` statement. Based on the particular resources received through a policy, the application may then make decisions about how to arrange its work.

2.6 Nesting Semantics

Since the `require` statement on line 3 is in the context of the `require` statement on line 10, the request must be satisfied out of the resources listed there. This restriction on nested `require` statements is true whether or not the nested statement is in the same thread or a child thread spawned in the `require` statement.

Here we also note that the allocations granted are not additive. That is, once a particular amount of a resource has been allocated, a larger amount may not be requested until execution leaves the `require` statement for the original amount. For example, a thread may *not* request 0.25 utilization with one `require` statement, and then 0.5 with a nested `require` statement. When this does occur, we consider it a programming error, and the program aborts with an error message[1]. Other features of Poli-C make this situation easy to avoid. For example, resource requests can also be phrased as a percentage of whatever is currently available. That is, one could write `require(cpuUtil(0,50%))` to obtain exclusive access to half of the existing allocation of utilization on core 0. On the other hand, a nested `require` statement may make requests for some utilization less that 0.25. In that case, the nested block of code is governed by the lesser amount for the nested `require` statement until execution leaves it.

2.7 Blocking

Only two cores are available after line 10, but an arbitrary number of threads may be spawned. Since each thread then proceeds to request exclusive access to one core, the result is that, unless `n` is 2 or less, some threads will block waiting for either core 0 or 1 to be available, and therefore only two threads will run at the same time, on separate cores. When the resources listed in a `require` statement are not available, execution blocks until they are. To prevent deadlock, when a thread blocks for any reason other than I/O, it gives up its claim to the resources it had already acquired, and attempts to reacquire them when unblocked.

It is also important to note that for a nested `require` statement to succeed, two things must be true. First, there must be enough of the resource available in the context, and second, the thread making the nested request must be the only running (non-blocked) thread sharing the resources granted by the outer `require` statement. This restriction is necessary so that there is no need to revoke resources from a thread. That is, we assume that threads within a single application behave cooperatively. Therefore, in this example, the spawned threads block after reaching their own `require` statements until the parent thread reaches the call to `join_all()`, where it blocks. It signals its blocked children, which wake up and acquire cores one-by-one.

3. Design Decisions

Now that we have presented a brief overview of Poli-C's features, we can discuss some of the design choices we made in building it. These include the choice between implementing Poli-C as a language extension or a library, the system-wide usage model we target, the meaning of a resource allocation, the behavior of threads when blocking, and restrictions on the flow of resources among threads.

3.1 Language Extensions vs. A New Library

Many of the features of Poli-C could be implemented either as language extensions or as part of a well-designed library. Both approaches have advantages and disadvantages. In general, while implementing new features as a library may be possible, we must also consider the design principle that a library which proves very difficult to use correctly would be

[1] We treat similarly cases where a program attempts to access an undefined device, resource kind, or policy.

better off as part of the language. In this way the use of its features can be rendered correct-by-construction, thereby reducing programming errors.

Researchers have argued that this is true of a number of existing libraries, e.g., pthreads [11], LRVM [41], explicit locking [21], and others. LRVM and explicit locking have the common trait that they require a user to make a call at the beginning of a section of code, and another call at the end of a section of code. However, it can be difficult to remember to place the right calls in the right places, to place only those calls actually needed, and to place the ending/closing call at all of the exits from a block of code. Poli-C is similarly structured; resource requirements have a beginning and an end. Both the language extension approach and the library approach must cope with this problem somehow.

A library-based implementation, to its credit, would require no custom compiler support, but two problems remain. First, we must ensure that the calls to the library that begin and end a resource requirement are matched correctly. Second, we must generate code to implement our policy DSL. Both of these problems could be addressed by clever uses of C++ lambdas, templates, operator overloading, and destructors on stack allocated variables. However, this would then preclude the use of Poli-C for pure C applications unless we also carried out substantial by-hand modifications of existing application code.

On the other hand, if the features of Poli-C are implemented through language extensions in a compiler, then the two problems mentioned above are solved by straightforward, compile-time code transformations, and Poli-C is still applicable to existing C code. However, the disadvantage of implementing Poli-C with language extensions is the need for this extra support from the compiler, which may complicate the tool-chain for an application's build, and therefore make the application harder to maintain and distribute.

In the design of Poli-C, we have chosen to modify the language instead of providing equivalent features in the form of a library of function calls. We have done this to retain support for existing C code. In particular, Poli-C requires only minimal changes to existing applications written in C. We must also justify our choice of C as our base language. Aside from the fact that there are many C programs that would benefit from Poli-C's features, language use and learning can be hindered by both excessive parsimony, as well as by a glut of features [33]. C, due its relatively simple features, is a good choice for trying out new language extensions. Even though we have made the choice to implement Poli-C with language extensions, we have made every attempt to keep the interface to our runtime library simple so that it can be easily integrated into other languages. Having made this choice for C, though, we mention that for C++, and possibly also other languages, a library-based implementation may be more appropriate.

3.2 Cooperation vs. Enforcement

Another important issue we consider is that of the model under which systems using Poli-C programs are shared. A design wishing to accommodate multiple competing users has different constraints than one wishing to accommodate a single user with competing applications. For us, this choice is reduced to the choice of who may supply the specialized operations used to define resource kinds. In particular, if the specialized operations do not correctly use OS features to mediate access to devices, then Poli-C will fail to provide the guarantees of the `require` statement.

Under the former sharing model, resource kind definitions should only be supplied or verified by the system administrator. However, under the latter sharing model, we must trust that the resource kinds defined by the single user behave properly. This makes sense because a single user derives no benefits from a misbehaving resource kind. If an *application* supplies a misbehaving resource kind, the system administrator or the single user may replace the specialized operations by modifying application source code, or if this is not available, by overriding the application versions with the dynamic linker.

We have have designed Poli-C with the latter sharing model in mind. However, the former could also be implemented by requiring elevated privileges to create new resource kinds, and by modifying `policd` to enforce per-user or per-group quotas.

3.3 Resource Allocations

It is often the case that a program can make-do with a range of resource levels. Therefore, in general, resource allocations come in two forms, a guarantee of a minimum level of resources or a limitation to some maximum level of resources. In Poli-C, the resources made available for use by a block of code through the `require` statement, what we call an allocation, are both a minimum level and a maximum level. At first glance, this may seem inflexible. However, Poli-C's policy functions allow resource levels to be chosen dynamically at runtime. For example, an application may provide a policy function parameterized by a range of acceptable resource levels. If the available resources are sufficient to satisfy some value in the range given to the policy function, then the block of code under the `require` statement receives the guarantee and is subject to the limitation, for which a value was chosen dynamically by the application's policy function. Through our application case-studies, we have found that our policies provide the flexibility needed to allow real-world applications to adapt to varying levels of resource availability and contention.

3.4 Blocking Behavior

Since our `require` statement is not additive, and since threads block at a `require` statement until sufficient resources are available, whether already acquired resources are retained or given up when a thread blocks is an important decision. If threads retain resources on blocking, then deadlock is possible if there is a waits-for cycle. Some such cycles can be prevented at compile time through static analysis. However, even without considering our policy functions, preventing these deadlocks would require a quite sophisticated flow-sensitive, whole-program analysis. Such a cycle can also be avoided by enforcing an order in which resources must be requested. However, this violates our composability design goal. In particular, an application developer should be free to use the `require` statement without worrying about what `require` statements will be attempted by library code.

Although potential deadlocks are eliminated, other problems remain if threads temporarily release resources on blocking. In particular, a thread may acquire exclusive access to a device using a `require` statement with the intention that no other thread should have access to it, only to unknowingly give up exclusive access by blocking in a library call. Additionally, since our runtime library must take action when a thread blocks, we must be able to identify every action a thread might take that could be construed as blocking. Since our current approach is to use the dynamic linker to override blocking functions in the C Library, we will miss, e.g. threads that busy-wait, or which access blocking system calls directly through any sort of `syscall` instruction offered by an ISA. At present, we regard the issues presented by this second approach to be less problematic; in fact they were no obstacle to applying Poli-C to the applications we examine in Section 7.

In the future, we believe this issue can be addressed by alerting the programmer to the existence of blocking functions inside of `require` statements at compile time, and by allowing the programmer to declare on *which* blocking operations resources should be released.

3.5 Hierarchical Parallelism

Threads may "grant" resources to child threads by spawning them inside of a `require` statement. The child threads may then never exceed the resources they receive in this way; they run inside of the parent's `require` statement for their entire lifetime. This design precludes, for example, the following situation. Thread 1 receives an allocation of resources through a `require` statement and spawns off some number of threads to work as a thread-pool. Thread 2 (not a member of the thread-pool) receives an allocation of resources through a `require` statement, and wishes to have the thread-pool started by Thread 1 perform some work on its behalf. Under our design, it is not possible for Thread 2 to grant its resources to the thread-pool in addition to the resources granted by Thread 1. In particular, our design constrains resources to be granted only from parent thread to child thread. Resources may not be granted across the thread hierarchy. This restriction simplifies our implementation, and has not created any problems in our application benchmarks. However, in the future, it may be worthwhile to create an abstraction for thread-pools that is understood by our language runtime.

C Statements	$stmt$	$::=$	$\ldots \mid$ **require**(rl) $stmt$
Request List	rl	$::=$	$rdv, rl \mid pdv, rl \mid \epsilon$
Resource Spec	$rspec$	$::=$	$r(ol, e) \mid r(d, ol, e)$
Policy Spec	$pspec$	$::=$	$p(ol, el) \mid lv = p(ol, el)$
Option	o	$\in$	{**ForChild**, **ForDescendants**, **BeforeSpawn**, **AfterSpawn**, **Private**, **Shared**}
Option List	ol	$::=$	$o, ol \mid \epsilon$
Expression List	el	$::=$	$e, el \mid \epsilon$
	$e,\ d$	$::=$	$CExps \mid percents$
	lv	$::=$	$Clvals$
	r, p	$\in$	$Identifiers$

Figure 2. Syntax of our C extensions.

4. Language Extensions

In this section, we describe the syntax of the `require` statement. Since we have already described the semantics of the `require` statement in Sections 2 and 3, in the rest of this section, we focus on the interface provided by Poli-C for defining resource kinds, and the domain-specific language for defining policies (e.g. the `cores` policy used in Section 2) for allocating sets of resources based on resource availability and contention at runtime.

4.1 Syntax

Figure 2 shows our addition to C's syntax. We add the `require` statement to the usual set of C statements. The `require` statement takes as an argument a non-empty list of requests, *rl*. The request list may contain both resource requests, *rspec*, and requests based on policies, *pspec*. The result of the policy function may be assigned to a C lvalue. The resources for the `require` statement may be parameterized by a C expression specifying the particular device, *d*, of that resource kind, *r*. For example, if the resource kind were CPU utilization, disk bandwidth, or network bandwidth, this parameter would specify the CPU ID, the block device, or the network interface, respectively.

The resources in the list are also parameterized by a possibly empty list of options, *ol*, and an expression specifying the requested amount of the resource, *e*, which may be either a C expression or a constant percentage, e.g. 50%. The options list can be used to change the timing and ownership of the resource allocation from the default. The meaning of these will be explained in Section 5. The type of the expression for the amount of the resource requested depends on the resource. However, the request may be phrased as a raw amount, or as a percentage of the total currently available.

```
void register_resource(char *name,
                       int n, char **devNames,
                       value_t *mins, value_t *maxs,
                       struct resource_ops *ops);

struct resource_ops {
  uint64_t (*get1)(int d);
  uint64_t (*get2)(int d);
  double (*calc)(int d, uint64_t t,
                 uint64_t ini1, uint64_t ini2,
                 uint64_t cur1, uint64_t cur2);
  void (*throttle)(int d, value_t *v);
  void (*restrict)(int d, int grant_deny);
};
```

Figure 3. Interface for adding custom resource kinds.

Policies are also parameterized by a list of options, as well as a list of arguments that are passed to the policy functions.

4.2 Resource Kind Definition

Resource kinds may be registered with our runtime using a library call, called `register_resource`. As mentioned earlier, resource kind definitions are a trusted component of Poli-C, and must maintain certain invariants, which we describe below. Subsequent to their registration, resource kinds may be referred to by name in `require` statements.

4.2.1 `register_resource`

The interface to resource registration is the function `register_resource` whose prototype is given in Figure 3. It takes the following arguments.

- `name` — The name of the resource kind (e.g. `cpuUtil`) used to refer to it in `require` statements.
- `n` — The number of devices of the resource kind.
- `devNames` — An optional array of strings specifying the names of the devices (e.g. "8:0" for the `major:minor` identifier for a block device under Linux, or "eth0" for a network device).
- `mins` — An array of values indicating the amount of the resource on each device shared by processes that have not yet used a `require` statement to obtain any of it. For example, this parameter can be used to constrain all processes that do not explicitly request the `cpuUtil` resource with a `require` statement to (part of) a single core.
- `maxs` — An array of values indicating the maximum allocation supported by each device.
- `ops` — A pointer to a structure that contains five function pointers defining the specialized operations for a resource kind.

The `value_t` type is a tagged union with fields of type `char*`, `uint64_t` and `double` so that the minimum and maximum values can be appropriate for the particular device. In a `require` statement, devices of the resource kind may be referred to either by an integer giving the index into the array of devices, or by one of the string identifiers in the `names` array. We assume that the information required to define resource kinds can be gathered by a program using features of the OS.

4.2.2 `resource_ops`

The functions of `resource_ops`, each of which is parameterized by a particular device, have the following meanings. The `get1` and `get2` functions are used to probe the device for cumulative usage statistics. For example, in the case of I/O device bandwidth, `get1` may return the number of bytes written. Usually only `get1` is defined, but `get2` may also be needed when a ratio is being measured. For example, calculating the cache-miss rate involves calculating the ratio of last-level cache misses to cache references. The `calc` function may be used to calculate a usage rate. Given a duration of time in nanoseconds, `t`, and the results of `get1` and `get2` at the beginning and end of the time interval, the rate of usage of the resource can be calculated. However, this is not required; `calc` may produce whatever value is meaningful for the device.

The `throttle` and `restrict` functions make use of any available OS features for limiting a thread's use of a device. The `throttle` function may wrap an OS function for limiting a thread to using a device at a particular rate given by the parameter `v`. Depending on the `grant_deny` parameter, the `restrict` function grants or denies access by the thread to a device. We distinguish between these two cases for the following reason. If the `throttle` function is not supplied, any throttling needed for the device (e.g. if a thread has `required` only a fraction of the resources available from a device) is supplied by a monitor thread that is part of our runtime, which periodically interrupts the thread and causes it to be blocked until its resource usage matches the prescribed rate. However, if a thread has only ever `required` 0% or a 100% of a device, the monitor thread does not run. Therefore, for resource kinds without a `throttle` function, the job of the `restrict` function, is to maintain the invariant that the OS does not allow a thread to use a device on which it has a 0% allocation. For example, when a thread requests a 0% allocation on a device of resource kind `cpuUtil`, the `restrict` function ensures that the core is removed from the thread's CPU set.

4.2.3 Resource Kind Initialization

When a program starts, it queries `policd` for the resource kinds that have already been defined. If it requires additional resource kinds, it gathers the necessary information from the OS, and calls `register_resource`. Then, `register_resource` sends the definition of the new resource kind to `policd`, which proceeds to track the resource kind for the entire system. When a new resource kind is registered using the `register_resource` function (or received from `policd`), the resource kind definition is placed in a hash table keyed by the resource name. Then, when a `require` statement mentions the name of the resource kind, the name is used to look up the definition in the hash table.

An advantage of this design is that porting our language extensions to a new OS or architecture is simply a matter of reimplementing the functions in `resource_ops` for each resource kind. When the use of our extensions does not affect the semantics of a program, only the performance, it is safe to move an application to a new platform on which only a subset of the resource kinds it refers to are defined.

4.2.4 Examples

Consider the `cpuUtil` resource kind used in the initial example. On Linux, for this resource, we define the `get1`, `calc`, and `restrict` fields of `resource_ops`. `get1` returns the total time spent by a thread on a CPU. `calc` calculates the percent utilization of a core by a thread, and `restrict` adds or removes the core from a thread's CPU set. Because the `throttle` field is left undefined, a monitor thread from our runtime is responsible for keeping the thread from using more than its allocation[2].

Many of the common hardware devices fit this way of looking at resources. On Linux, the `cgroups` [2, 34] interface allows fine-grained control over the share of disk and network utilization each group may use. For memory, system calls exist to keep particular pages from specific banks resident in memory. For caches, various patches to Linux implement a page-coloring VM subsystem [42] that expose an interface that could be used here to grant threads exclusive access to parts of caches, however we have not yet implemented this.

It is also possible to use this formulation of resources to act as a traditional semaphore or monitor. That is, instead of allocating concrete hardware resources, resource kind definitions can be written to mediate access to abstract software resources, for example OS resources like file descriptors, or user-level resources like simultaneous access to a particular software module. In Section 7 we demonstrate one such example of the use of these "virtual" resources.

4.3 Policies

As mentioned earlier, requesting access to amounts of resources based on an explicitly given list of devices is neither convenient nor portable. On the other hand, it is useful to make requests, not only for some amount of a particular device, but also to make requests for sets of allocations from groups of devices depending on availability and contention. For example, a high-level task may require two cores, but may or may not care which cores they are. Therefore, we include in Poli-C a way of specifying these sorts of requests, which we call *policies*. Policies are defined using a simple, C-embedded DSL we have developed for the purpose. Finally, unlike resource kinds, policies are *not* trusted. That is, new policies can be declared and used by unprivileged users.

$$
\begin{array}{lcl}
\mathit{CDecls} & ::= & \ldots \\
 & | & \textbf{policy}\ \mathit{id}(\textbf{own\_t o},\ \textbf{void}\ *\textbf{out}, \ldots) \\
 & & \quad \{\mathit{Cstmts}\} \\
\mathit{Cstmts} & ::= & \ldots \\
 & | & \textbf{res\_iter}(\mathit{rid})\ \mathit{Cstmts} \\
 & | & \textbf{dev\_iter}(r,\ \mathit{did})\ \mathit{Cstmts} \\
\mathit{Clvals} & ::= & \ldots \mid d.\mathit{dfield} \\
\mathit{Cexps} & ::= & \ldots \mid \textbf{dev}(r, d) \mid \textbf{dev\_count}(r) \\
 & | & \textbf{max\_available\_dev}(r) \\
 & | & \textbf{min\_waiters\_dev}(r) \\
 & | & \textbf{total\_waiters}() \\
\mathit{dfield} & \in & \{\textbf{id}, \textbf{available},\ \textbf{request},\ \textbf{ownership}, \\
 & & \ \textbf{waiters}\} \\
r, d & ::= & \mathit{Cexps} \\
\mathit{rid}, \mathit{did} & \in & \mathit{Identifiers}
\end{array}
$$

Figure 4. Syntax of our policy definition DSL.

4.3.1 Syntax

Figure 4 shows the syntax of our policy DSL. We add to C an additional declaration for defining policy functions. The first two formal parameters to the policy function are mandatory. The first, `o`, gives the requested resulting ownership of the resource set to be computed by the policy function, i.e. Shared, Private, or Default. The meanings of these ownership options are described in Section 5. The second argument to the policy function, `out`, allows the policy to pass policy-specific information about the results of the policy function back to the application, for example through the binding for `c` in `require(c=cores(n))`. The remaining arguments are the policy-specific parameters provided by the application through the `require` statement (e.g. the `n` of `cores(n)`).

Inside of a policy definition, two statements are offered in addition to the usual C statements: one for iterating through resource kinds (`res_iter`), and the second for iterating over the devices of a resource (`dev_iter`). Inside of the block of code under `res_iter(r)`, `r` is a pointer to the name of the resource.

In `dev_iter(r,d)`, `r` specifies the name of the resources whose devices will be iterated over. Inside the block under the statement, `d` is bound to an instance of a *device structure* that contains the following fields. The `id` field gives either the string that was supplied for the device name if there was one, or a numeric id otherwise. The `available` field gives the amount of available resource on the device. The `request` field may be written by the policy function to indicate how much of the resource should be requested from the device by the policy function. The `ownership` field may be written by the policy function to indicated the desired ownership for the requested amount. Finally, the `waiters` field gives

[2] On Solaris, a call could be supplied here for the `throttle` function—in particular one that uses the Fair Share Scheduling feature of its Zones [3].

the number of threads waiting for resources on the device to come available.

Additionally, a number of interesting functions are made available by our compiler inside of policy definitions. The `dev(r,d)` function returns an instance of the device structure described above, given a string constant for the resource `r`, and a device identifier `d`. The device identifier can be either a string constant for the name of the device, or an integer giving the index for the device. We supply the `dev()` function so that a policy author can find the device structures without using our iteration functions. The `dev_count()` function returns the number of devices for a resource kind. The functions `max_available_dev()` and `min_waiters_dev()` return the device of a resource kind with the most resource available and the fewest waiters, respectively. Finally, the function `total_waiters()` returns the total number of threads waiting for resources across the entire system.

4.3.2 Semantics

As the policy definition executes, it reads the `available` and `waiters` fields, and writes the `request` and `ownership` fields of various device structures. Writing the fields of device structures either bound by the `dev_iter` statement or obtained through the `dev` function is meaningful. In particular, those amounts of those devices of those resource kinds will be requested by the policy function. If the policy cannot be satisfied by the resources it finds in the device structures, the policy function must return `PolicyFailure`. Otherwise, the policy function must return `PolicySuccess`.

The effects of `require`-ing a policy are as follows. Atomically, our runtime attempts to execute the effects of a `require` statement for each of the device structures that were written by the policy function. If the policy function returns `PolicyFailure`, or if one of these implied `require` statements fails, then the executing thread must give up its resources and block, re-executing the policy function before retrying the implied `require` statements.

Further, the resource allocations obtained by a thread through a policy are treated as a unit. They are allocated as a unit, deallocated as a unit, and temporarily given up as a unit when threads block. That is, normally an allocation is temporarily returned if none of the threads sharing it are currently running. However, if an allocation was obtained as part of a policy, it is only returned if no threads sharing *any* of the allocations obtained for the policy are currently running.

Figure 5 shows how the `cores` policy is defined using our policy DSL. It simply iterates through the available cores, looking for those on which 100% utilization is available. It is easy to see how this policy could be modified to achieve a number of different policies, for example allocating some minimum number of cores, and returning the resulting core map in the `out` parameter.

As with resource kind registration, when our compiler front-end finds a function matching the right signature for

```
typedef struct {
  int ncores;
} cores_t;

policy cores(own_t o, void *out, int n) {
  int i, found = 0;
  cores_t *c = (cores_t *)out;

  c->bits = 0;
  dev_iter ("cpuUtil", d) {
     if (d.available == 1.0 && found < n) {
       d.request = 1.0; d.ownership = o;
       if (c) c->ncores++;
       found++;
     }
     else {
       d.request = 0.0; d.ownership = o;
     }
  }

  if (found < n) return PolicyFailure;
  return PolicySuccess;
}
```

Figure 5. The definition of our `cores` policy in terms of the basic resource cpuUtil.

a policy definition, a pointer to the policy function is placed in a hash table keyed by the policy name. When a `require` statement mentions a policy name, the name is used to look up the policy function in the hash table.

5. Options for Composability

We have implemented a number options that modify the `require` statement's default behavior. In this section, we take a look at an example that demonstrates Poli-C's ability to compose parallel libraries. In particular, we will use a few of Poli-C's options to the `require` statement to cope with unavailable code and idiosyncrasies of a commonly used parallel library. In this example, a program divides a workload into several partitions, and spawns a thread to work on each partition. Each of these threads then attempts to work on pieces of each partition in parallel. The structure of this example is analogous to the structure of the scientific application we use to evaluate Poli-C in Section 7.

5.1 Without Poli-C

Figure 6(a) shows the organization of this program without the use of Poli-C or the parallel library (in this example we'll use OpenMP). Instead of C, we use pseudocode here in order to focus on the structure of the program. The `main` function uses the `partition_calc` function to partition the workload `C` into a number of pieces after examining the number of cores, defined by the parameter `NCORES`. Then for each of the partitions, the `main` function spawns a thread to run the function `do_part`. The `do_part` function iterates through the

```
void *do_task(task_t *t):
  task(t);

void *do_part(part_t *P):
  foreach t in P:
     spawn(do_task, t);

int main():
  calc_t *C;
  partition_calc(C,NCORES);
  foreach p in C:
    spawn(do_part, p);
```

(a) No Poli-C, No OpenMP

```
void *do_task(task_t *t):
  require(cores(1)):
    task(t);

void *do_part(part_t *P):
  require(cores(P.cnt)):
    foreach t in P:
       spawn(do_task, t);

int main():
  calc_t *C; cores_t c;
  require(c = min_cores(1)):
    partition_calc(C, c.ncores);
    foreach p in C:
      spawn(do_part, p);
```

(b) With Poli-C, No OpenMP

```
void *do_part(part_t *P):
  require(cores(BeforeSpawn, P.cnt)):
    omp_set_num_threads(P.cnt);
    require(cores(AfterSpawn,
                  Private, 1),
            cores(ForChild, 1)):
      omp_calc(C);

int main():
  calc_t *C; cores_t c;
  require(c = min_cores(1)):
    partition_calc(C, c.ncores);
    foreach p in C:
      spawn(do_part, p);
```

(c) With Poli-C, With OpenMP

Figure 6. Example program with Poli-C working with OpenMP.

tasks that must be performed for the partition P, and spawns a thread to run each in parallel.

This design has two main problems. First, in such an application NCORES is usually provided by the user or programmer as either a pre-processor macro, an environment variable, a command-line argument, or by making the right system call. This is a problem because there is no straightforward way for the application to know at this stage what other applications are running on the system, and to adjust the partitioning of C accordingly. Secondly, the OS scheduler is unaware that the tasks spawned to work on the same partition are cooperating with each other and competing for resources with the threads for other partitions. In particular, the threads for each partition may wish to meet up at barriers, or may rely on sophisticated cache optimizations. Competing with the threads of other partitions increases the wait time at barriers, and pollutes a thread's cache.

5.2 With Poli-C

Figure 6(b) shows how the `require` statement of Poli-C can be used in the program. In the `main` function, we use the policy `min_cores`. This policy attempts to acquire exclusive access to as many cores as possible, with a minimum given by the argument to the policy. If the minimum is not available the program blocks. When more cores come available, the program is signaled by `policd`, and executes the policy function again.

The result of the policy function is assigned to `c`. Its field `ncores` gives the number of cores acquired in the `require` statement. Now, the calculation `C` can be partitioned based on the actual number of uncontended cores that the program is guaranteed access to. As before, one thread is spawned for each partition. Each of these threads runs the `do_part` function. This function first uses the `require` statement to gain exclusive access to a number of cores equal to the number of tasks in the partition[3]. Since the tasks for each partition will have exclusive access to a core for each task, we avoid the problems with barriers and caches suffered by the original program. Finally, in `do_task` a `require` statement is used to acquire exclusive access to one core, which has the effect of pinning the thread executing `task(t)` to a single core. This is likely unnecessary; since the core is uncontended, the Linux scheduler will avoid migrating the thread to a different core, but we `require` the core anyway just to be sure.

5.3 Poli-C with GNU OpenMP

Finally, in Figure 6(c), we consider what happens when, instead of `do_part` manually spawning threads and managing the tasks, the program uses OpenMP to process each partition. For this example, we consider the GNU implementation of OpenMP, in which a new thread-pool is spawned for each new thread that begins an OpenMP job. Given that, without the Poli-C features that we describe below, the same problems would exist here as with the code in Figure 6(a).

In this configuration, the `main` function is as before. The `do_part` function, however, is substantially different because we use options to the `require` statement to manage OpenMP. In particular, it is not incorrect to do so, but we would like to avoid making another request for a single core with the `require` statement in `do_part` if only a single core was obtained with the `require` statement in `main`. Therefore, in the first `require` statement in `do_part`, we use the `BeforeSpawn` option. It ensures that the stated resources or policies won't be put in place until just before the first thread is spawned inside of the `require` statement. GNU OpenMP does not spawn a thread-pool if it is instructed to use only one thread. Thus, the `require` statement is only triggered if OpenMP is to use multiple threads.

[3] We assume that it is an invariant of `partition_calc` that the number of tasks per partition will not be greater than its second argument, which specifies the number of cores.

Option	Description
`ForChild`	Directs spawned threads to make a resource request
`ForDescendants`	Directs all descendant threads to make a resource request
`BeforeSpawn`	Delays effects until just before a child thread is spawned
`AfterSpawn`	Delays effects until just after a child thread is spawned
`Private`	Not shared with spawned threads
`Shared`	Shared with anyone

Table 1. Options for the `require` statement.

Now, as in Figure 6(b), we would like to pin each OpenMP thread to its own core. Unfortunately, we no longer have access to a convenient place to put the default `require` statement, as the OpenMP threads are spawned from library code that we do not have access to. However, we can use options to the `require` statement to achieve the same effect. In the second `require` statement in `do_part`, there are two policies. In the first policy we use the options `AfterSpawn` and `Private`. Here, we use `AfterSpawn` so that the `require` statement won't take effect until just after the first thread is spawned inside of it. We use the `Private` option so that the parent thread can retain exclusive access to one of the cores allocated by the first `require` statement. The net result is that just after spawning the first thread of the OpenMP thread-pool, the parent thread will get exclusive access to a single core that is not available for allocation to any of the thread-pool threads. Finally, in the second policy in the second `require` statement, we use the `ForChild` option. This simply directs any child thread spawned in the `require` statements to apply the `cores` policy just after they begin. This second `require` statement has the net effect of pinning each of the OpenMP threads to its own core.

Here we must note that a different OpenMP implementation may have different idiosyncrasies, and therefore require a different arrangement of `require` statements and options. Although this situation is not ideal, we feel that it is more portable than the currently available alternatives, i.e. relying on by-hand configuration on each platform the code runs on, or creating custom builds of the different OpenMP implementations, as Lithe would require, for example.

5.4 Additional Options

This example has shown how the use of Poli-C can avoid the performance problems that may happen when parallel libraries are composed. The `require` statement accepts a few other options. Their names and effects are listed in Figure 1. Of these, the `Shared` option possibly requires some explanation. A thread that uses the `Shared` option in a `require` statement is subject to the resource limitation for the requested allocation, but receives no guarantee of exclusivity. That is, the allocation may also be shared with any other thread that requests the resource with `Shared` ownership.

```
// Basic resource kind before translation
require(cpuUtil(0,100%)) {
  ...
}
```

```
// Basic resource kind after translation
require_push("cpuUtil",0,Default,PC,1.0);
...
require_pop();
```

```
// Policy before translation
require(c = cores(2)) {
  ...
}
```

```
// Policy after translation
policy_push("cores", Default, &c, n);
...
pop_policy();
```

Figure 7. Poli-C syntax before and after translation.

6. Implementation

The implementation of Poli-C has three parts. First, the language extensions are implemented using a source-to-source compiler for C. Second, within a process, among threads, the semantics of Poli-C are enforced by a runtime library that tracks resource allocations using an *allocation tree*. Finally, we have implemented a system-level daemon, `policd`, to enforce the semantics of Poli-C among all processes running on a system. In this section, we describe each of these parts in turn.

6.1 Language Extensions

Our extensions to C are implemented in about 1700 lines of OCaml using the CIL source-to-source compiler [35]. The Poli-C compiler front-end has two primary functions. First, it must translate our `require` statement into calls into our language runtime. Second, it must identify policy functions, compile them to C, and register them with our language runtime.

6.1.1 require compilation

We explain the translation of the `require` statement by way of two example statements, one using a resource kind and one using a policy. Consider the code fragments in Figure 7. In the first two fragments, the program requests 100% of device 0 of resource kind `cpuUtil` using the default semantics. This is translated into the `require_push` call in the second fragment. Here, `cpuUtil` is given as a string, the device is specified as an integer, the options for the require statement are passed as an enum value (in this case `Default`), and the value requested is specified with a floating point value, with the code `PC`, which indicates the value is a percent.

In the third and fourth fragments, the translation is performed for a policy. Our compiler can identify a request for a policy because there must exist in the program a function with the same name as the policy. In the third fragment, the program requests any two cores using the default options. This is translated into the call to the `policy_push` call in the fourth fragment. Here, the name of the policy is given as a string, there are no options, the binding for the result of the policy function is made by passing the address of `c`, and the arguments to the policy function follow.

6.1.2 Policy function compilation

Since there may be an arbitrary number of arguments to a policy function, `policy_push` is a variable argument function. Furthermore, since the call to the policy function may need to be delayed if the BeforeSpawn, AfterSpawn, ForChild, or ForDescendant options are used in the `require` statement, the arguments to a policy function may need to be saved for later. To accomplish this, for each policy function, our compiler generates helper functions to save and restore the arguments. When a policy is registered with the Poli-C runtime by adding it to a hashtable keyed by the policy name, not only do we add a pointer to the policy function itself, but also to the helper functions. This way, `policy_push` can look up all of the functions it may need.

Policy functions are identified by the compiler as any function having a compatible type, i.e. the return type is the enum type for the `PolicyFailure` and `PolicySuccess` return codes, and the first two argument types are as described in Section 4.3. Two elements of our policy DSL require translation. First, the `res_iter` and `dev_iter` are compiled to simple `for`-loops. Second, writes to the `request` and `ownership` fields of device structures are compiled to calls to functions that store the written values into a table. If the policy function returns `PolicySuccess`, the values from this table indicate what resources to request.

6.2 Language Runtime

The runtime for our extensions is implemented in about 11k lines of C, of which about 2400 lines are the Linux-specific resource kind definitions. The runtime operates by using the dynamic linker to intercept calls into the pthreads library along with selected system calls. In particular, the runtime must take action when threads are created, blocked, and destroyed.

In order to implement the semantics described in Section 2, for each resource kind and device, each thread maintains a stack that mirrors the nesting structure of the `require` statements, which we call a *resource stack*. Furthermore, since threads inherit the allocations of the threads that spawned them, and these allocations may outlive the duration of the `require` statement in the spawning thread, for each resource kind and device, we must also maintain a reference counted tree of *allocations*, which we call an *allocation tree*. In order to explain the operation of our language

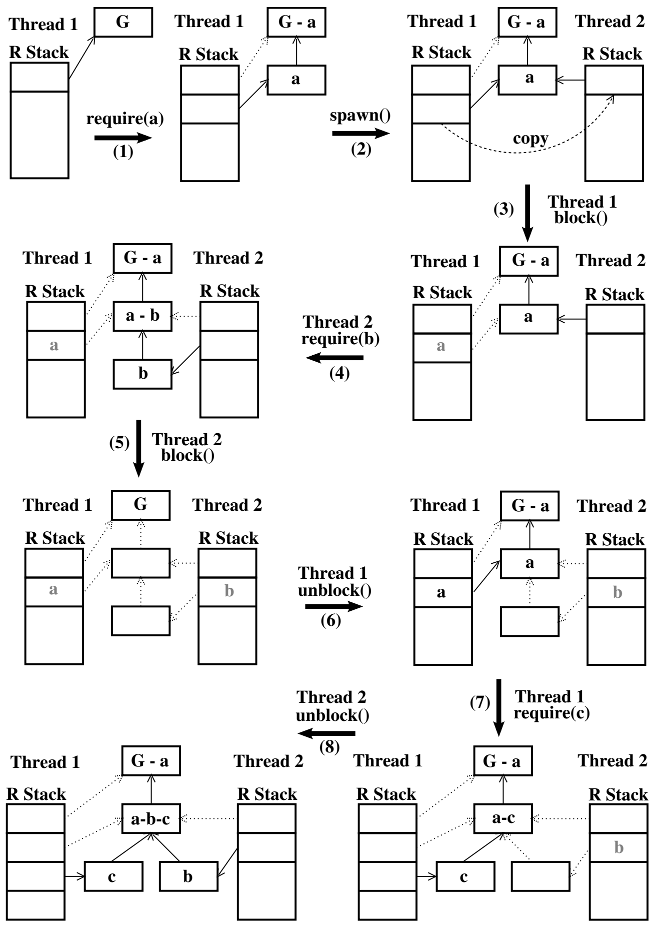

Figure 8. Starting at top-left, actions taken on the stacks and allocation tree for entering `require` statements and spawning threads.

runtime, we must explain what happens to a thread's resource stacks, and allocation trees when various operations occur, in particular: entrance into a `require` statement, exit from a `require` statement, thread blocking, thread spawning, and thread exiting.

We explain most of these operations by way of the example diagram in Figure 6.2. The diagram shows the modifications made to the stacks and allocation tree for a single resource `R` by the above operations. In the diagram, thick, sold arrows labeled by threads and actions indicate state transitions. The parenthesized numbers near these arrows are labels that we use to refer to the transitions in the text. The thin dotted and non-dotted arrows indicate different kinds of references among resource stack elements and allocation tree nodes. The distinction between dotted and non-dotted reference will be described below.

Transition 1: The initial state is in the upper-left corner. Thread 1's `R` resource stack contains one element pointing to the root node of the allocation tree for `R`, which contains the maximum available amount of the resource, `G`. If `G` is found to be 0, and the root node has no children, then the runtime queries `policd` for resource availability. In the first step, Thread 1 enters a `require` statement, making a request for the amount `a` of `R`. Since we will suppose that `G` is large enough to accommodate the request for `a`, in response, our runtime library pushes an element onto Thread 1's resource stack that contains a pointer to a new allocation tree node. The new allocation tree node contains the value of the new allocation. Its parent is the root of the allocation tree. Since `a` of the resource is held in the new node, the root node's value is decremented by `a`. Further, if resource kind `R` has a `throttle` or `restriction` function defined, it is called for Thread 1 now.

Reference Counting: Notice also that the reference from the original resource stack element to the root node of the allocation tree is now represented by a dotted arrow. This indicates that the allocation in that node is no longer "used" by the thread. An arrow may become dotted either, as in this case, when it comes from a non-top element of the resource stack, when it does come from the top element but the thread is blocked, or when it comes from an allocation node with no incoming non-dotted arrows. Further when an allocation node has no incoming non-dotted arrows, it returns its allocation to its parent node. In summary, a reference represented by a dotted arrow indicates that the allocation node is not currently being used by the holder of the reference, but that it may do so again at some point in the future.

Transition 2: In the second step of the diagram, Thread 1 spawns a thread. Since Thread 2 is spawned inside of the `require` statement initiated by Thread 1, Thread 2 shares the allocation. This is indicated by copying the top entry from Thread 1's stack to be the first entry on Thread 2's stack. Further, `throttle` and `restriction` functions are called for Thread 2 using `a` when it begins running.

Transition 3 & 4: In the next step, Thread 1 blocks. It remembers the value of its allocation, and switches its reference from a non-dotted to a dotted arrow. Since the allocation node still has a non-dotted arrow, it does not return the allocation to its parent; it is still being used by Thread 2. Next, Thread 2 enters its own `require` statement for an amount `b`. Since we suppose that `a` is sufficient to accommodate the allocation, in response the runtime library pushes a new element on Thread 2's resource stack, and creates a new allocation tree node, just as it did for Thread 1's `require` statement.

Transition 5: Following Thread 2's `require` statement, it blocks. Since Thread 2 was the only thread using its most recent allocation, the allocation node returns the value `b` to its parent, and changes its reference to its parent from non-dotted to dotted. Now, however, the parent node also no longer has any incoming non-dotted references. This makes sense since neither Thread 1 nor Thread 2 is running. Therefore, its value is also returned to its parent, and so the root allocation node returns to its original value `G`.

Transition 6 & 7: Once Thread 2 has blocked, Thread 1 unblocks. Using the remembered value from the top of its stack, through the allocation tree node referenced there, it reacquires the allocation from its initial `require` statement. Following this, Thread 1 attempts a second `require` statement for the amount `c`. Since we suppose that `a` is sufficient to accommodate the allocation, a new stack element, and allocation tree node are created as usual.

Transition 8: Then, Thread 2 is unblocked. Two things could happen at this point. First, if `a-c` is less than `b`, then Thread 2 must block. When blocking due to insufficient resources, a thread blocks on a condition variable held in the first allocation tree node it encounters during a traversal toward the root that has incoming non-dotted references, but which fails to satisfy the request. In this case this would be the allocation tree node holding `a-c`. However in our diagram, we suppose that `a-c` is large enough to accommodate `b`, and the state is updated accordingly.

Epilogue: To finish our example, although it is not pictured in our diagram, when a thread leaves a `require` statement, it pops off its topmost stack element. If the allocation tree node it refers to no long has any references, its allocation is returned to its parent, and the node is deallocated. This process repeats recursively up the tree. If a thread exits before leaving a `require` statement, its resource stacks are emptied following this same procedure.

Options and Policices: We must mention a few more details that were not covered by the above example. First, the timing options to the `require` statement are implemented by maintaining multiple stacks for each resource kind and device, one for each of the options. For the ownership options, allocation tree nodes maintain reference counts and flags to distinguish among, and keep state for, Private, Shared, and Default allocations. Second, as mentioned above, allocations received by way of a policy are treated as a unit. This is accomplished by placing each allocation for a policy in a linked list. When deciding whether or not to return an allocation to the parent node, we simply traverse the list, checking reference counts. Finally, we note that our runtime library is protected by a big global lock. In applying Poli-C to highly parallel applications, this did not cause any scaling problems, however it would not be difficult to construct artificial examples in which it would. We leave elimination of this potential bottleneck for future work.

6.3 `policd`

The above description of our runtime system is effective at maintaining the invariants of Poli-C only within a single multi-threaded process. To enforce these invariants across an entire system, we use a system-level daemon process called `policd`. `policd` has two key functions. First, it performs bookkeeping for the resource kinds. Second, it monitors the system for process creation and exit events in order to maintain Poli-C invariants. We use `policd` to make up for the absence on Linux of an existing, query-able service that tracks resource allocations. If an OS already has such a service, then `policd` would likely be unnecessary.

6.3.1 Bookkeeping

Programs using Poli-C communicate with `policd` over a local, UNIX socket using the `PASSCRED` option to allow message receivers to verify the identity senders. When a Poli-C program begins, it requests a list of the available resources from the daemon. If it defines some resource that does not exist, it is registered with the daemon by sending a description of the resource over the socket. Then, the daemon tracks how much of the resource is distributed to each process running in the system. When a program executes a `require` statement, if it has an insufficient amount in the root node of its local allocation tree, it makes a request to the daemon. If an insufficient amount is available, the program requests that the daemon notify it when the amount changes, and blocks on a `select` call on the socket. When the root node of the allocation tree has no children, the amount of the allocation is returned to the daemon. To aid in writing policy functions, a process can also request that the daemon send it the number of threads currently awaiting a notification about resources from the daemon.

6.3.2 Monitoring

When a new resource kind is created, a minimum level of the resource is specified on each device. This minimum level is shared among processes that have made no resource requests. By constraining processes that have made no requests to a limited set of resources, we are able to make guarantees to those that have made requests. To enforce this minimum level, `policd` subscribes to events emitted by Linux when processes are forked and when they exit. When `policd` starts, every existing process is made to share the minimum resource level[4], as is each new process that is forked. When a process is allocated resources through the `require` statement, it no longer uses the minimum amount that was set aside. Further if a process exits without returning its resources, the daemon notices the exit event and reclaims them.

7. Evaluation

We evaluate Poli-C by looking at the ease with which it enables performance improvements. We have made small changes to a few applications, which we treat below as case studies. First, we look at a state-of-the art implementation of sparse matrix QR decomposition, which is used in a number of different scientific and financial applications. Further, we use Poli-C to improve the performance of three different web services using the Apache http server. Finally, we apply Poli-C to a state-of-the-art implementation of multicore database join algorithms. We ran all of our experiments on a large server machine: a 4-socket, 64-core AMD Opteron 6276 running at 2.3GHz having 256GB of main memory in four banks and a 2-disk RAID array running Debian Linux with a 3.2.2 kernel. The topology of the machine is such that each NUMA-node, as reported by `libnuma`, consists of 8 cores, which have private L1 caches, and shared L2 and L3 caches.

[4] This not possible e.g. for Linux kernel-mode threads that are bound to specific resources when spawned by the kernel.

7.1 QR Decomposition

We applied Poli-C to a state-of-the-art implementation of QR decomposition [14]. QR decomposition is used in many important applications for linear, least-squares fitting. The algorithm used in this implementation creates a tree of sub-matrices, each of which may be processed in parallel. Parallel linear algebra operations may then be used to operate on each of these sub-matrices. Similar strategies are also beginning to be used for other matrix factorization algorithms [31].

Creation of, and operation over, the tree of sub-matrices is handled by Intel's TBB library [38]. Then, the linear algebra operations are performed over the sub-matrices by BLAS matrix subroutines [16]. Any implementation of the BLAS subroutines can be linked with the application. We used the Intel MKL [25], which uses the GNU OpenMP library to parallelize the matrix operations.

By default, the number of threads used by TBB, and the number of threads used by OpenMP are both chosen to be equal to the number of cores in the system. This is problematic, however, because Linux will cause different OpenMP tasks working on different sub-matrices to share the same core. This results in poor performance as threads pollute the caches of other threads. Further, there are many additional context-switches, and teams of cooperating OpenMP tasks fail to arrive at barriers at the same time.

To avoid these problems, an end-user would have to configure TBB and OpenMP by hand to set the number of threads they use to values that prevent this interference. Further, even if these parameters are set to good values, two problems still remain. First, there is still no assurance that the kernel will not schedule threads in a way that causes undesirable interference. Second, this by-hand configuration must be redone on each new platform. In our view, an end-user is unlikely to attempt either an initial good configuration, or any reconfiguration for a new platform, especially if the QR decomposition library is only called deep within an application.

7.1.1 Poli-C Policy

Any automatic approach that prevents the OpenMP jobs from interfering with each other would go a long way towards solving the performance problem described above. We would like an automatic approach that requires no end-user intervention, but we are willing to tolerate changes to a few lines of code.

To that end, we devise a Poli-C policy, called `numa_nodes`. The policy is very much like the `cores` policy described in

Matrix Method	LLC Misses	Ctxt Switch	CPU Switch	Time(s)
landmark				
BASELINE	$9.0x10^7$	$4.1x10^5$	$9.3x10^4$	19.97
BYHAND	$2.1x10^7$	$8.2x10^3$	$2.0x10^2$	2.65
POLI-C	$8.1x10^6$	$3.3x10^3$	$2.3x10^2$	**2.64**
deltaX				
BASELINE	$2.3x10^9$	$9.9x10^6$	$2.3x10^6$	188.94
BYHAND	$2.0x10^8$	$1.1x10^4$	$1.6x10^2$	8.08
POLI-C	$1.2x10^8$	$6.0x10^3$	$2.3x10^2$	**7.26**
ESOC				
BASELINE	–	–	–	> 1000
BYHAND	$7.3x10^8$	$3.4x10^4$	$1.7x10^2$	**42.01**
POLI-C	$4.2x10^8$	$2.5x10^4$	$2.0x10^2$	42.51

Figure 9. Summary of results for the QR decomposition benchmark.

Section 4.3 with the exception that instead of attempting to acquire exclusive access to some number of cores, it uses `libnuma` to attempt to acquire exclusive access to some number of NUMA-nodes.

Applying Poli-C to the QR decomposition for the management of cores requires only the addition of 2 `require` statements. In particular, the `require` statements we added when using the `numa_nodes` policy follow the structure laid out in the discussion of options to the `require` statement in Section 5.

7.1.2 Results

We ran the QR decomposition on three large sparse matrices taken from the MatrixMarket [12] maintained by the US National Institute of Standards and Technology. We ran the QR decomposition on matrices using the default configuration with default parameters (BASELINE), the default configuration with hand-tuned parameters (BYHAND), and Poli-C.

The table in Figure 9 shows the results of running the QR decomposition. We used the matrices “landmark”, “deltaX” and “ESOC” as our workloads. For each of the methods and column, the table lists the average of 10 runs. The standard deviation was never bigger than 10% and usually under 1%, so we omit the exact values.

For each matrix, the first line shows the out-of-the-box performance of the workload using the default options for the numbers of TBB and OpenMP threads (BASELINE). The second line gives the best performance obtained when the numbers of TBB and OpenMP threads were set by hand (BYHAND). To arrive at the parameters for the BYHAND configuration, we did not conduct an exhaustive search. Instead, we took the suggestion of the developers of the QR decomposition and chose the parameters so that their product would equal the number of cores. For these experiments we chose 4 TBB threads and 16 OpenMP threads. The third line gives the performance of Poli-C.

7.1.3 Discussion

On our 64-core machine, the default configuration performs quite poorly. Communication and thread creation overhead, in addition to the interferences described above, degrade performance to the extent that we saw no point in waiting for the benchmark to finish running on the ESOC matrix. Clearly, without some tool to help the programmer, by-hand tuning would be necessary for this library to be of any use on a large machine. We have found that our `numa_nodes` policy performs as well as by-hand tuning on this machine. In order to achieve this on a range of machines, it may be necessary to devise a more sophisticated policy. One of the key advantages of the design of Poli-C is that this policy information is transparent and well-organized through the use of our policy DSL. In particular, the author of a library like this QR decomposition could develop a suggested policy to distribute alongside the library itself.

7.2 Image manipulation web server

In this case study we investigate a web server that manipulates client-uploaded images. The first manipulation supported by the server is an image blur, consisting of one relatively expensive operation. The second manipulation generates several different resizings of an image, consisting of multiple relatively inexpensive operations.

We use the Apache web server [4], the cgic CGI library [5], and the ImageMagick image manipulation library [6], which uses OpenMP to parallelize the image operations. In its default configuration on Debian, Apache creates a number of server processes each of which spawns a number of threads for handling client requests. Therefore, in this case study, we rely on `policd` to coordinate resource management among the separate processes.

By default, each of the OpenMP jobs would attempt to use all of the cores. When the webserver is handling more than one client at a time, this arrangement creates the same problem that existed with the QR decomposition. The form of the solution in this case is similar, however there is one important difference: the web server must perform as well as possible under many different loads. That is, unlike the QR decomposition, we have no control over how many concurrent OpenMP jobs there will be. This difference motivates us to use a new Poli-C policy.

7.2.1 Poli-C Policy

We experiment with two different policies. In the first policy, cores are divided fairly among concurrently arriving requests. For example, if there are 4 concurrent requests, and 64 available cores, then each request will be allocated 16 cores. This policy is called `fair_cores`. The second policy mirrors the policy for the QR decomposition above. It attempts to allocate exclusive access to a NUMA-node to each request. However, if the number of cores divided by the number of concurrent requests is smaller than the number of

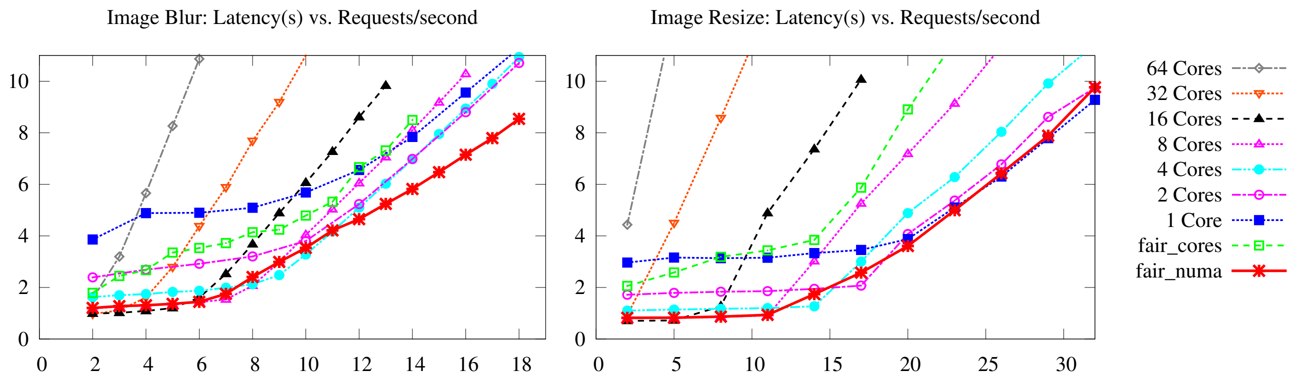

Figure 10. Average latency in a 10 second burst versus request frequency for our image blurring and resizing webserver.

cores in a NUMA-node, then each request will attempt to acquire exclusive access to that number of cores on the same NUMA node instead (with a minimum of one core). This policy is called `fair_numa`.

7.2.2 Results

Our benchmark consists of ten second bursts of requests. Client and server are on the same LAN so that essentially all of the measured latency is due to time the server spends working. We vary the number of requests per second, and measure the average latency over all requests in the burst. We do this experiment using an image file of constant size (about 3MB). The constant size benefits less adaptive policies that always use the same number of cores for each request.

Figure 10 shows the results of the benchmark using a number of different configurations. Each plotted point is the average of ten runs. Standard deviations were between 1 and 5%, so we omit error bars. The "*n* Cores" configurations do not use Poli-C, and simply use *n* cores to process each blur request. Initially, for large *n* this provides good performance. However, as request frequency, and thus contention for cores, increases, performance drops off rapidly. When the specified *n* is smaller, performance degrades more gracefully. At high request rates, the adaptive `fair_numa` policy used by Poli-C performs as well as or outperforms the other methods. The `fair_cores` policy eventually outperforms *n* Cores for $n \in \{64, 32, 16\}$.

7.2.3 Discussion

There are a few interesting features of this experiment. First, we note that the default configuration that uses all 64 cores again performs quite poorly. When using all cores, communication costs drown out the performance benefit of increased parallelism, even when the request rate is low. Performance degrades from there as contention increases.

At various request frequencies, various of the "*n* Cores" methods outperform Poli-C. However, with the exception of the fixed single core policy at high frequency[5], no fixed policy is consistently better. This implies that core allocation policies must provide a mechanism for adaptation to achieve consistent performance. The language extensions provided by Poli-C are an easy way to do this. Finally, we note that `fair_numa` outperforms `fair_cores` largely thanks to its NUMA-awareness.

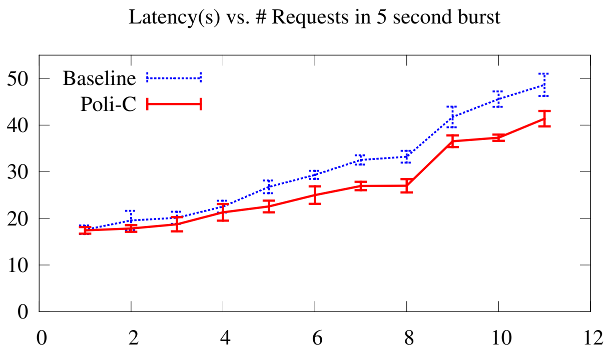

Figure 11. Average latency versus the number of uploads in a five second burst for our file upload server.

7.3 File-upload web server

We now consider a case study in which a webserver accepts uploaded files, and in order to provide a high degree of assurance to clients, does not send a response until the uploaded files are safely in place on permanent storage. In particular, when our CGI script receives an upload, it writes it to a file using the `O_SYNC` and `O_DIRECT` flags of the `open` system call. Even with a RAID array that increases write bandwidth, we find that performance degrades when many clients are concurrently writing large files in this way.

Using the Linux `cgroups` subsystem, we can devise a policy that guarantees (and limits) each client a portion of

[5] Since the individual resize operations are small, the difference between Poli-C and the single core policy is essentially measuring the overhead of the Poli-C runtime.

disk utilization. In particular, the `blkio.weight_device` controller of a `cgroup` can be written with a value from 10 - 1000 signifying the share of disk utilization that each `cgroup` may use, with the sum of shares for all `cgroups` in the system being limited to 1000. We abstract this mechanism by creating a `DiskWeight` Poli-C resource kind. Using this resource kind, we can guarantee to each file-upload client exclusive access to the disk in turn. In the CGI script, we simply wrap the `write` system call in one of our `require` statements using a request for `DiskWrite(1000)`.

We evaluated this policy by uploading a 200MB file from an increasing number of clients. The destination file and directory of each upload was unique to avoid effects unrelated to contention for the disk. Figure 11 shows the results of this experiment both with and without Poli-C. Each data point shows the average of 10 runs. Error bars show standard deviation. At all request frequencies, Poli-C provides a performance improvement between 1 and 20% increasing as request frequency increases.

7.4 Multicore Database Join

In this final case study, we show how Poli-C may be used to improve the performance of multicore database join algorithms. Database systems use a number of different join algorithms depending on the needs of the overall query and the relations they are joining. Different join algorithms have different cache locality properties. The work of Lee et. al shows that scheduling a join algorithm with strong cache locality on the same cores as an algorithm with weak cache locality leads to poor performance [28], but that scheduling strong with strong, or weak with weak, tends to have a smaller performance penalty. Lee et. al solve this problem through modifications to the query scheduler of a database system, and through changes to the underlying OS's virtual memory subsystem. We will address the problem in this case study by devising a Poli-C policy to meet the same scheduling constraints.

In particular, we examine a highly optimized parallel radix partition join algorithm, which has strong cache locality, and a parallel join algorithm that uses a no-partitioning approach, which has weak cache locality [6].

7.4.1 Poli-C Policy

To implement the strategy suggested by Lee et. al, we introduce two "virtual" resource kinds that we use in our policy to keep track of which cores are currently being used by a strong join algorithm, and which are being used by a weak join algorithm. We call these resource kinds virtual because they are simply used for bookkeeping rather than being attached to actual hardware. The resource kinds are called `StrongCore` and `WeakCore`, and each has as many devices as there are cores on a particular machine. Anticipating that the policy we define will allocate to each join algorithm

[6] We thank Cagri Balkesen for the implementation of these algorithms.

```
void join(join_t *j) {
  struct numa_out out;
  if (j->type == Radix) {
    require(out = strong_numa_node()) {
       radix_join(j,out.ncores);
  }}
  else {
    require(out = weak_numa_node()) {
      nopartition_join(j,out.ncores);
}}}
```

Figure 12. Application of Poli-C to multicore join algorithms using the `strong_num_node` and `weak_numa_node` policies.

an amount of 1.0 for each virtual core it will use, we set the maximum available amount for the virtual cores to be an integer multiple of 1.0 indicating the maximum number of concurrent join algorithms that may share a core. In our experiments we chose 2.0, indicating that at most two concurrent join algorithms could share a core.

We defined two Poli-C policies, `weak_numa_node` and `strong_numa_node`. These policies are used in the join algorithm driver using our `require` statement as indicated in Figure 12. The `strong_numa_node` policy attempts to construct a resource request that meets the following constraints:

- Using Poli-C's `Shared` option, Request each core in a NUMA-node, preferring first nodes that have not been allocated, and then the least-full node that already has a `StrongCore` allocation.
- If the policy function notes that it is the first to request a particular node, it not only requests a unit (1.0) of the `StrongCore` virtual resource, but also *all* of the `WeakCore` virtual resource. This prevents a weak join algorithm from running on the same core.
- Request an allocation of 0.0 on all other cores.

The implementation of `weak_numa_node` is symmetric with `strong_numa_node`.

7.4.2 Results

Before proceeding to implement the above policies, we first sought to verify the premise on which they are based. In particular, we performed the following experiment. We ran two instances of the parallel radix join algorithm concurrently on the same NUMA node, followed by two instances of the parallel no-partition join algorithm, followed by one instance of each. Each instance operated on separate pairs of relations, one small and one big. The small relation consisted of roughly 16 million tuples occupying 128MiB. The large relation consisted of roughly 256 million tuples occupying 2GiB. Figure 13 shows the results of this experiment. Running the radix join(strong) concurrently with the no-partition join(weak) harms the performance of the radix join, and slightly helps the performance of the no-partition join.

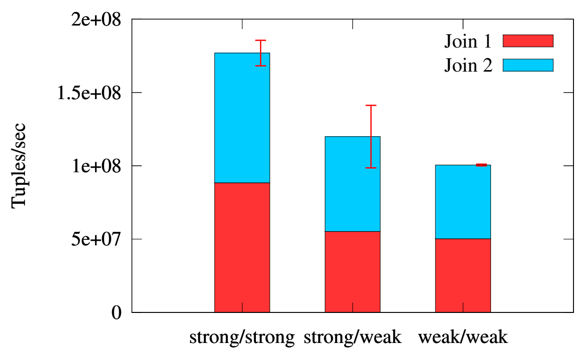

Figure 13. Throughput in tuples per second while concurrently running two different database join algorithms with the indicated cache locality properties.

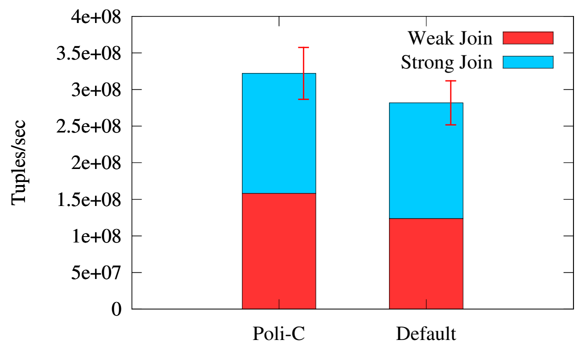

Figure 14. Throughput in tuples per second for two different methods of mapping multicore join algorithms to cores.

Having verified that there is indeed a performance benefit (at least for the radix join algorithm) in avoiding sharing cores among different join algorithms, we now proceed to an experiment that shows the performance benefits of the policies described above. In particular, we ran 16 separate joins using the relations described above. 8 used the radix join algorithm, and 8 used the no-partition join algorithm. The results of this experiment are shown in Figure 14. When using Poli-C, the average throughput was about 322 million tuples joined per second. When allowing join algorithms to be mapped to cores arbitrarily, the average throughput was about 282 million. Thus, Poli-C enabled a 14% advantage in throughput. This advantage is not as large as it could be. A more sophisticated policy could take advantage of many of the strategies suggested for successful database/OS co-design [20]. We leave implementation of these strategies in the context of Poli-C for future work.

8. Related Work

The most closely related work to our own is the hierarchical scheduling library, Lithe [36]. Lithe is used to efficiently compose multiple parallel libraries in the same application. While Lithe requires no annotations in application code, it does require modification of, and custom builds of, the parallel libraries that it composes. Given the preponderance of parallel libraries (e.g. OpenMP, pthreads, TBB, etc.), their implementations (e.g. GNU, Intel, Microsoft, Sun), and the different versions relied upon by various applications, we view it as less intrusive to allow a detailed resource allocation policy to be declared in the application source code, and to implement the runtime library for these annotations once-and-for-all, using features provided already by operating systems for manipulating thread scheduling and resource allocation.

On the other hand, the use of custom libraries also has advantages. In Poli-C we have tried to provide features sufficient to handle the unavailability of library code in the form of our options to the `require` statement. However, it may still be the case that the best place for one of our `require` statements is not available to the application programmer. A runtime that takes advantage of custom parallel libraries avoids this problem because a call to the right scheduling primitive can be placed anywhere.

The second major difference with Lithe is that Poli-C is able to manage not only cores, but also any resource whose usage can be measured, or to which the OS exposes an interface. We accomplish this by decoupling resource allocation from thread scheduling (or stacks and continuations as in Lithe) through the use of our explicit allocation trees, which we describe in Section 6.

Finally, Lithe is only able to manage cores within a single multithreaded process. This implies that it would not have been able to act as Poli-C did in the webserver case study above. In particular, our system-level daemon `policd` allows resource coordination and polices to operate among several multithreaded processes.

Aside from Lithe, researchers have proposed a number of other hierarchical scheduling solutions for general-purpose computing, as well as real-time scheduling. These include Psyche [32], Converse [26], CPU Inheritance [18], HLS [37], GHC [29], and Manticore [17]. The largest difference between these approaches and our own is the ability of our language extensions to manage not only cores, but also other system resources. Additionally, our goal was not enabling the implementation of hierarchical schedulers, but rather, among other things, enabling the efficient composition of existing parallel libraries and programs through statements declaring explicit resource management policies.

Many recently developed languages and language extensions have sought to ease the expression of parallelism and concurrency, for example Cilk [19], Jade [39], Go [1], Split-C [27], Fortress [7], X10 [15], Chapel [13], and AC [22]. Our work differs from these projects in that our language extensions allocate resources to various parallel tasks, taking for grated that many languages already have features for the expression of parallelism. Since the API for our language runtime is simple, it should be easy for Poli-C to be integrated with these languages.

Finally, OS resource management features have also recently been used to support virtualization [23, 24]; guest OSs are limited to using only part of the machine, as specified by the system administrator, so as not to interfere with other guest OSs or applications running on the host OS. Because these features are becoming more generally useful, Poli-C gives applications more direct access to them.

9. Conclusion and Future Work

In this paper, we presented Poli-C, a language-based interface for the composition of parallel software and the fine-grained, hierarchical management of resources. Our design gathers disjoint and confusing OS APIs into a single coherent interface, and we have demonstrated that it improves the performance of a number of representative benchmarks on a modern server machine using only a few additions to application code.

At present, our implementation is only for Linux, however all platform specific code is wrapped up in our resource kind definitions. We hope that this will make porting Poli-C to other platforms straightforward. In particular it would be very interesting to implement a port to Barrelfish [9], Tesselation [30], FOS [43], or another next-generation, multicore OS in order to investigate possible synergies between OS-level, and user-level scheduling and resource management.

We are investigating ways of making our policy DSL more declarative. Our current approach requires looping through each device of each desired resource kind. This becomes cumbersome when the requirements of a policy become more complex, for example when requesting that allocated cores be on the same NUMA node, as in our case studies in Section 7. We believe that asking programmers for declarative constraints, and then using an SMT or constraint logic programing solver to solve for a set of resources that satisfy the constraints will be a more convenient approach.

10. Acknowledgements

Many thanks are due to Tim Harris, Kornilios Kourtis, and Timothy Roscoe for discussions and advice about early drafts of this paper, and to Cagri Balkesen for help in setting up the database joins benchmark. We would also like to thank the anonymous reviewers for their valuable feedback and suggestions.

References

[1] The go programming language, Oct. 2011. http://golang.org/.

[2] lxc: Linux containers, Sept. 2011. http://lxc.sf.net/.

[3] System administration guide: Oracle solaris containers-resource management and oracle solaris zones, Sept. 2011. http://docs.sun.com/app/docs/doc/817-1592.

[4] Apache HTTP server project, Apr. 2012. http://httpd.apache.org/.

[5] cgic: an ANSI C library for CGI programming, Apr. 2012. http://www.boutell.com/cgic/.

[6] ImageMagick: convert, edit, and compose images, Apr. 2012. http://www.imagemagick.org.

[7] Allen, E., Chase, D., Luchangco, V., Jr., J.-W. M. S. R. G. L. S., and Tobin-Hochstadt, S. The fortress language specification version 1.0, 2008. http://research.sun.com/projects/plrg/fortress.pdf.

[8] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. The landscape of parallel computing research: A view from berkeley. Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[9] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The multikernel: a new os architecture for scalable multicore systems. In *SOSP'09*, pp. 29–44.

[10] Bienia, C., Kumar, S., Singh, J. P., and Li, K. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (October 2008).

[11] Boehm, H.-J. Threads cannot be implemented as a library. In *PLDI'05*, pp. 261–268.

[12] Boisvert, R. F., Pozo, R., Remington, K., Barrett, R. F., and Dongarra, J. J. Matrix market: a web resource for test matrix collections. In *Proceedings of the IFIP TC2/WG2.5 working conference on Quality of numerical software: assessment and enhancement* (London, UK, UK, 1997), Chapman & Hall, Ltd., pp. 125–137.

[13] Chamberlain, B., Callahan, D., and Zima, H. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl. 21*, 3 (2007), 291–312.

[14] Davis, T. A. Algorithm 915, SuiteSparseQR: Multifrontal multithreaded rank-revealing sparse QR factorization. *ACM Transactions on Mathematical Software 38*, 1 (2011).

[15] Ebcioglu, K., Saraswat, V., and Sarkar, V. X10: Programming for hierarchical parallelism and non-uniform data access. In *OOPSLA'04*.

[16] et al., C. L. Basic linear algebra subprograms for FORTRAN. In *Transactions on Mathematical Software* (1979).

[17] Fluet, M., Rainey, M., and Reppy, J. A scheduling framework for general-purpose parallel languages. In *ICFP'08*, pp. 241–252.

[18] Ford, B., and Susarla, S. Cpu inheritance scheduling. In *OSDI'96*, pp. 91–105.

[19] Frigo, M. Multithreaded programming in cilk. In *Proceedings of the 2007 international workshop on Parallel symbolic computation* (2007), pp. 13–14.

[20] Giceva, J. Database-operating system co-design. Master's thesis, ETH Zürich, May 2011.

[21] Grossman, D. The transactional memory / garbage collection analogy. In *OOPSLA'07*, pp. 695–706.

[22] Harris, T., Abadi, M., Isaacs, R., and McIlroy, R. AC: Composable asynchronous io for native languages. In *OOPSLA'11*.

[23] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A., Shenker, S., and Stoica, I. Nexus: A common substrate for cluster computing. Tech. Rep. UCB/EECS-2009-158, EECS Department, University of California, Berkeley, 2009.

[24] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. Mesos: a platform for fine-grained resource sharing in the data center. In *NSDI'11*, pp. 22–22.

[25] Intel. Math kernel library for the linux operating system: User's guide, 2007.

[26] Kalé, L. V., Yelon, J., and Knuff, T. Threads for interoperable parallel programming. In *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing* (1997), pp. 534–552.

[27] Krishnamurthy, A., Culler, D. E., Dusseau, A., Goldstein, S. C., Lumetta, S., von Eicken, T., and Yelick, K. Parallel Programming in Split-C. In *SUPERCOM'93*, pp. 262–273.

[28] Lee, R., Ding, X., Chen, F., Lu, Q., and Zhang, X. Mcc-db: minimizing cache conflicts in multi-core processors for databases. *Proc. VLDB Endow. 2*, 1 (Aug. 2009), 373–384.

[29] Li, P., Marlow, S., Peyton Jones, S., and Tolmach, A. Lightweight concurrency primitives for ghc. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop* (2007), pp. 107–118.

[30] Liu, R., Klues, K., Bird, S., Hofmeyr, S., Asanović, K., and Kubiatowicz, J. Tessellation: space-time partitioning in a manycore client os. In *HotPar'09*, pp. 10–10.

[31] Mackey, L., Talwalkar, A., and Jordan, M. Divide-and-conquer matrix factorization. In *Neural Information Processing Systems (NIPS)* (2011).

[32] Marsh, B. D., Scott, M. L., LeBlanc, T. J., and Markatos, E. P. First-class user-level threads. In *SOSP'91*, pp. 110–121.

[33] McIver, L., and Conway, D. Seven deadly sins of introductory programming language design. In *Proceedings of the 1996 International Conference on Software Engineering: Education and Practice (SE:EP '96)*, pp. 309–.

[34] Menage, P. Cgroups, July 2011. http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

[35] Necula, G. C., McPeak, S., and Weimer, W. CIL: Intermediate language and tools for the analysis of C programs. In *CC'04*, pp. 213–228. http://cil.sourceforge.net/.

[36] Pan, H., Hindman, B., and Asanović, K. Composing parallel software efficiently with lithe. In *PLDI'10*, pp. 376–387.

[37] Regehr, J., and Stankovic, J. A. Hls: A framework for composing soft real-time schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium* (2001), pp. 3–.

[38] Reinders, J. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* O'Reilly, 2007.

[39] Rinard, M. C., and Lam, M. S. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst. 20*, 3 (1998), 483–545.

[40] Saha, B., Adl-Tabatabai, A.-R., Ghuloum, A., Rajagopalan, M., Hudson, R. L., Petersen, L., Menon, V., Murphy, B., Shpeisman, T., Sprangle, E., Rohillah, A., Carmean, D., and Fang, J. Enabling scalability and performance in a large scale CMP environment. In *EuroSys'07*, pp. 73–86.

[41] Satyanarayanan, M., Mashburn, H. H., Kumar, P., Steere, D. C., and Kistler, J. J. Lightweight recoverable virtual memory. In *SOSP'93*, pp. 146–160.

[42] Taylor, G., Davies, P., and Farmwald, M. The tlb slice: a low-cost high-speed address translation mechanism. In *Proceedings of the 17th annual international symposium on Computer Architecture* (New York, NY, USA, 1990), ISCA '90, ACM, pp. 355–363.

[43] Wentzlaff, D., and Agarwal, A. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev. 43* (April 2009), 76–85.