Inflation and Deflation of Self-Adaptive Applications

Ryan W. Moore
University of Pittsburgh
Pittsburgh, PA 15217 USA
rmoore@cs.pitt.edu

Bruce R. Childers
University of Pittsburgh
Pittsburgh, PA 15217 USA
childers@cs.pitt.edu

ABSTRACT

Autonomic multicore systems dynamically adapt themselves in response to run-time conditions and information for a variety of purposes, such as fault tolerance, power conservation, and performance balancing. Multiple application processes must coordinate their efforts and share resources to achieve system goals. In this paper, we present our inflate/deflate programming model for building autonomic processes and systems. The inflate/deflate programming model provides application-specific knowledge and reactions to a central resource coordinator. The central resource coordinator distributes and revokes resources at runtime to achieve a system goal. We discuss the overall design and challenges involved in our model. We test our design for adaptable programs by modifying programs from the PARSEC benchmark suite. The programs are tested in two sample situations to explore the difficulties of modification and the rewards gained. We find that the first modified program (blackscholes) fairly shares CPU time with other system workloads in an energy conservation scenario (up to 50% more efficient than an unmodified blackscholes). The second modified program (dedup) dynamically takes advantage of core resources as they become available (17% faster performance). If no new cores become available, it is able to more efficiently use existing resources (9% faster performance).

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*; C.1.3 [**Processor Architectures**]: Other Architecture Styles

General Terms

Systems, multicore computing, adaptation

Keywords

Multicore computing, autonomic systems, run-time heterogeneity, power management, run-time adaptation

SEAMS '11, May 23-24, 2011, Waikiki, Honolulu, HI, USA

1. INTRODUCTION

Applications that execute on multicore systems experience heterogeneity. This nonuniform environment may change at run-time unpredictably. Today's sources of heterogeneity include NUMA-based architectures, program phase behavior, competition between threads for CPU time, thermal throttling, and others.

Future applications on future multicore systems are expected to be subject to current sources of heterogeneity as well as additional sources. Cores might be designed to feature different capabilities or may be subject to aging, where components malfunction or perform less efficiently than they originally performed.

Furthermore, multicore systems, whether present or future, have diverse goals. Possible goals include maintaining a particular quality of service, maximizing raw program throughput, performing energy management, avoiding thermal emergencies, or computing within an energy budget. System goals may even be composed. For example, the system may need to maintain a particular quality of service without causing a thermal emergency.

Applications must appropriately respond to heterogeneity if they are to meet their goals. If an application does not respond to heterogeneity, poor performance, or poor energy savings may result. For example, if a core's functional units slow down, the core may become unable to calculate floating point results as quickly as it previously could. In response, the application might choose to no longer use that core and instead use another. If the application does not respond, it is possible that it may not meet performance goals.

To allow applications to respond to heterogeneity, we propose a new programming paradigm known as inflate/deflate. Inflation of an application is when a resource is given to the application, increasing its total resources. Deflation is when a resource is given away by the application or taken away. Combined with system-level and application-level policies, inflation/deflation allows for dynamic adjustment to run-time heterogeneity, enabling the creation of autonomous processes and systems.

The specifics of how to create inflate-capable and deflate-capable programs, as well as their policies is currently unknown. Once an application is deployed and running, it should respond to a variety of sources of heterogeneity. Some sources of heterogeneity, however, may not have been anticipated by the application developer. Even if a source of heterogeneity was anticipated, the developer may be unable to fully reason about it, and thus, may be unable to create an application that always performs the proper adaptation.

Resource
core
functional unit
cache
memory (RAM)
peripherals (GPU, NIC, memory controller)

Figure 1: Inflate/deflate manageable resources

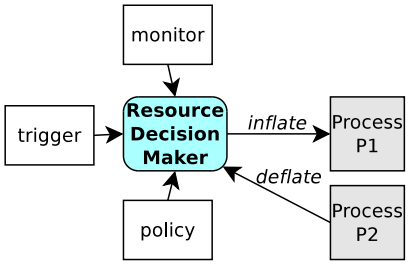


Figure 2: High level inflate/deflate model

For example, if a relatively slow core becomes available to the application, should a thread be migrated to this new core? Which one? Should a new thread be created? The proper response may be application-specific, input-specific, and/or system-specific. A balance must be found between developer effort and inflation/deflation effectiveness.

In this paper, we explore the above questions and experiment with the inflate/deflate model. In Section 2, we present the inflate/deflate model, discuss its purpose and how it works, present an example inflate/deflate-capable application, and identify concrete challenges as well as possible solutions. In Section 3 we conduct experiments on two PARSEC[2] programs using a partially complete inflate/deflate model, to determine the challenges and benefits of our model. Section 4 discusses related works and Section 5 concludes.

2. MODEL

We use the term "inflate" to describe when an application is given a resource that it previously did not possess. The term "deflate" refers to when an application no longer has an assigned resource. Deflation may occur either involuntarily or voluntarily. In this paper, our focus is inflating and deflating applications by giving and taking away cores. Figure 1 lists other possible resources that may be inflated or deflated. The concepts of inflation and deflation are a way to think about and create self-adapting applications (known as "inflate/deflate-capable" applications) and autonomic systems.

With inflation/deflation-capable applications, a component is required to manage resource allocation. We call this component the "Resource Decision Maker (RDM)." Figure 2 shows a sample interaction between this component and two inflate/deflate-capable applications.

The Resource Decision Maker receives input from a monitor, which measures the system's current and past behavior. On a trigger, based on the monitor's recorded information and the current policy, the Resource Decision Maker assigns a new resource to process P1. The Resource Decision Maker also retrieves a resource from process P2. The resource taken from P2 and given to P1 may be the same.

The Resource Decision Maker observes the application's utilization of a resource. If an application does not properly utilize a resource, the resource may be retrieved and perhaps given to another application. Resources may be voluntarily or forcefully returned. The Resource Decision Maker also makes predictions about its actions, described later in Section 2.2.

Application-level knowledge must be utilized to maximize the benefits that an inflation/deflation-capable application can provide. If an application is granted a resource, the application should take full advantage of the new resource. Similarly, applications should properly deal with the loss of a resource.

Consider a multi-threaded work-stealing application which at arbitrary points will be either granted a core or have a core taken away. Also, assume that the application is compute bound. The application should, upon receiving a core, create a thread to run on that core. If assigned enough cores, the application might change its primary algorithm. For example, an algorithm which scales well may be used if many cores are assigned to the application. If few cores are assigned to the application, an alternative algorithm may be used. Applications may have limits to how much parallelism their algorithms can exploit (e.g., due to lock contention). If an application is unable to utilize a resource, the resource should be released back to the Resource Decision Maker.

The Resource Decision Maker may also retrieve a core from an application (i.e., an application is forced to give up a core). As with granting cores to an application, this action should be coordinated with the application. The application may have to prepare itself for the removal of the core. Additional time may be necessary to ensure that the application's threads are always in an acceptable state. For example, a core's thread(s) may need additional time to finish and commit their work before the core can be returned.

The Resource Decision Maker should respect an application's minimum requirements. For example, an application may require at least two cores at all times to service outside requests. Other applications may have no such constraints.

Figure 3 lists some examples where inflate/deflate can be used to adapt execution in response to dynamic changes in cores. The figure also shows possible Resource Decision Maker responses. The Resource Decision Maker might give a core to a program (inflate) and/or take away a core from a program (deflate).

The first example is for a core failure. If a core fails, the Resource Decision Maker should force any process with threads on that core to move their threads away from the core (deflate). An invoked failure recovery mechanism may allow the application to recover.

The second and third example situations also require deflating. In the second example, the system's power consumption is too high. The Resource Decision Maker can force a decrease in overall CPU utilization by taking a core from an application (deflate). Similarly, if the system's programs are meeting their quality of service goals, it may be possible for the Resource Decision Maker to remove a core from an application. Removing a core from an application with quality of service requirements must be carefully done to ensure that the application will still meet its QoS requirements after the core is removed.

Situation	Inflate Core?	Deflate Core?
1. a core has completely failed	×	✓
2. power consumption rate is too high	×	✓
3. quality of service easily being met	×	✓
4. a core has come online	✓	×
5. an application has exited, freeing resources	✓	×
6. cores have differing capabilities	✓	✓
7. user priorities	✓	✓

Figure 3: Situations where inflate/deflate can be used for self-adapting applications

The fourth and fifth examples benefit from the Resource Decision Maker giving cores to programs (inflate). In the fourth example, a new core has come online. Applications may now use it. In the fifth example, an application has exited, thus freeing resources. These newly freed resources may now be used by other applications.

The sixth and seventh situations may require both inflation and deflation. In the sixth situation, the system's cores are heterogeneous. The threads in the system's applications may be more suited to some cores than others. Over time, the Resource Decision Maker learns information about the behavior and requirements of applications. Guided by this information, it may take away a core with certain characteristics from one application (deflate) to give the core to another application which can make better use of that core's characteristics (inflate). In the seventh situation, user priorities may inform core assignments. For example, if the system is idle except for one user's application, the Resource Decision Maker may give all cores to that application. If another user starts an application on the system, the Resource Decision Maker may take cores away from the existing application in order to give them to the new application.

Applications likely do not natively support adaptation. They are unable to respond to Resource Decision Maker requests. Yet, such functionality is necessary to allow the Resource Decision Maker to act appropriately. The application developer must therefore add such functionality. We discuss this issue next.

2.1 Programming Model

Applications must respond appropriately when granted or revoked resources. Figure 4 shows example pseudocode for what we expect application developers to write. Application developers possess application-level knowledge, which can be leveraged for self-adaptation. Programming language constructs, such as those proposed in the pseudocode, can allow developers to express this knowledge without undue burden.

Figure 4's pseudocode introduces two important concepts. First, the figure introduces the concept of a "sketch" (of resources). A sketch is a set of loose specifications about what resources are needed for a particular code section in an application. Secondly, the pseudocode describes the application's reaction to various requests or commands. We first discuss the sketch concept, then we discuss the reaction descriptions.

Application developers, being familiar with the application, have high-level algorithmic knowledge. For example, the developer may know that an application computes large arrays of floating point values. This suggests that the application would benefit from fast cores with large caches. Another application, such as a static content web server, may not benefit from large caches.

The developer may also know other useful, low-level information. For example, a pipelined scientific process, in addition to using floating point operations, periodically communicates between threads in adjacent stages (i.e., one stage's output is another stage's input). Here, the programmer has some insight about thread communication but may not know the precise communication patterns and behaviors that manifest at run-time.

One purpose of the inflate-deflate model is to allow the developer to easily and succinctly express high-level application knowledge to the Resource Decision Maker. By allowing the developer to easily express this knowledge, the system can more accurately adapt itself to meet run-time goals. Sketches, without being burdensome, allow the programmer to express their application knowledge. Application knowledge may also be obtainable offline from the compiler and/or profile information or learned online.

On line 2 of Figure 4, the application requests multiple fast cores (`FAST + MULTIPLE + CORE`), but additionally indicates that its request is lenient (`LENIENT`). This is an example of a sketch. In this example, the application is flexible in what type of core the application can run on. Slow cores may automatically be given to applications which state that they can use slow cores (e.g., I/O-bound applications). If no such applications exist, the figure's sample application will nevertheless be given a chance to acquire the slow core due to the `LENIENT` directive. The sketch informs the Resource Decision Maker's concrete resource assignment at runtime.

Applications may reason about resources other than cores, such as RAM usage or GPU access. This sample application cannot. Resources that match the sketch's description are granted by the Resource Decision Maker to the process (`requestFromSystem`). With its granted resources, the application performs its work (lines 4 - 21).

When utilizing cores, the "inParallel" construct is always the first construct executed. It is responsible for creating threads to work on the cores. Once all cores have a thread assigned to them, the application waits until its worker threads have signaled that all work is done (`waitForWorkDone`, line 7). When the system grants the application a core resource, the application responds by spawning a new thread on the newly assigned resource (`onInflate`, line 8). Similarly, when the application is told to return a CPU resource (`onDeflate`, line 11), it signals the thread assigned to that core to quit. The thread will soon shut itself down, freeing the core to be returned to the system (not shown).

The sample application in Figure 4 can also deal with core resource changes, albeit in a limited way. Resources may change as applications use them. These changes can be intentional, such as if the Resource Decision Maker forces a change, or unintentional, such as a speed decrease due to a

```
1  def main(argv):
2          resources = requestFromSystem(FAST + MULTIPLE + LENIENT + CORE)
3
4          inParallel {
5                  for resource in resources:
6                          spawnThread(workerThreadFunc, resource)
7                  waitForWorkDone()
8          } onInflate(CORE, newResources) {
9                  for resource in newResources:
10                         spawnThread(workerThreadFunc, resource)
11         } onDeflate(CORE, removedResources) {
12                 #Signal the thread to quit (freeing the resource) when convenient.
13                 for resource in removedResources:
14                         getThreadAssignedTo(resource).signalAssignedThread(SHUTDOWN)
15         } onDeflate(CONTENTION, coresWithContention) {
16                 if (CONTENTION.type == CACHE && CONTENTION.value >= HIGH):
17                         #Signal threads to change how they compute their results.
18                         for resource in coresWithContention:
19                                 thread = getThreadAssignedTo(resource)
20                                 thread.signalAssignedThread(USE_LOW_FOOTPRINT_ALGORITHM)
21         }
```

Figure 4: Example pseudocode for an adaptive application

local thermal emergency. One such unintentional change in one core may be from the behavior of other cores. This can occur, for example, due to cores sharing one or more levels of cache with neighboring cores. The neighbor cores' cache usage may create cache contention, negatively affecting the original core's ability to utilize the caches (e.g., neighbor cores may evict other core's data from the caches).

The program in Figure 4 can respond to changes in cache contention (`onDeflate`, line 15). The program checks to see if the core cache contention is high. The exact run-time meaning of "high cache contention" is determined by the Resource Decision Maker and depends on the system's cores and workload. If cache contention is high, the program signals its threads to reduce cache usage (line 20). Threads will receive the signal and utilize a different algorithm to reduce their cache footprint (not shown).

The difficulty of creating inflation/deflation applications which respond to system run-time changes depends on how the original application was written. Some existing applications are already written in ways which allow for inflation/deflation. For example, queues between pipeline stages allow worker threads to be oblivious about which threads produced their stage's input. Work stealing approaches (e.g., Intel® Thread Building Blocks[11]) further abstract the production of work and the assignment of threads to the jobs that consume the work.

2.2 Run-time System

The Resource Decision Maker makes better management decisions when it has detailed knowledge about the system, its applications, and its applications' behaviors. To allow the Resource Decision Maker to learn such knowledge and coordinate application actions, we envision the run-time support system shown in Figure 5. The figure also shows a sample interaction between the components in the support system.

In the figure, the run-time system is made aware of an external event which requires adaptation. The Resource Decision Maker considers the system's current state (e.g., operating system state, RAM, cores, temperatures, etc.). The Resource Decision Maker also communicates with processes in order to learn information about the processes' state and their capabilities. Processes are made up of their application

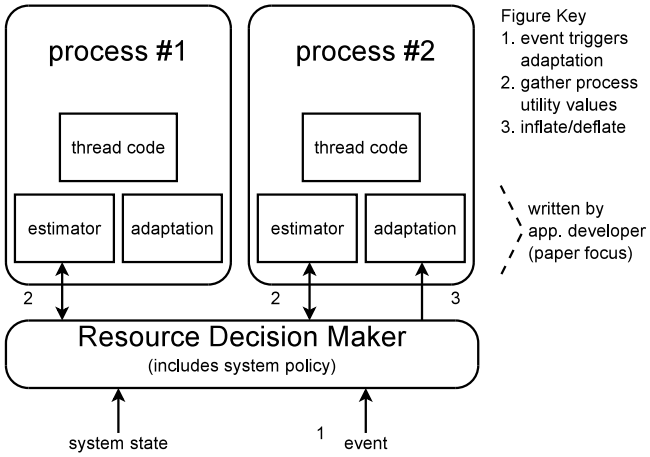


Figure 5: System adaptation architecture

code (thread code) along with code capable of estimating the process's ability to use resources (estimator code) and code that allows the process to respond to resources being given or taken away (adaptation code). The estimator code and adaptation code are expected to be expressed in a form similar to that seen in Figure 4.

When an event happens (1) the Resource Decision Maker responds appropriately using its system policy. For example, if a core becomes available for computation the Resource Decision Maker's policy may consider assigning the core to a process. The Resource Decision Maker communicates with each process instructing it to estimate how well it is able to use the newly available resource (2). The processes, after examining their current state (e.g., resource assignment, computation phase), reply back to the Resource Decision Maker with how well they are able to use the new resource (2). Sketches greatly inform this utility estimation.

The Resource Decision Maker's policy next decides which process should get the new core. This decision is not made entirely based on the process utility estimations. The Resource Decision Maker may consider other information (e.g., a policy which evenly distributes resources between users).

The Resource Decision Maker then gives the resource to a process. In the sample interaction in Figure 5, process #2 acquires the resource and invokes its adaptation code (3).

2.3 Model Challenges

There are several open, difficult questions posed by our vision of the inflate/deflate programming paradigm. In this section, we describe the challenges (with possible solutions) to guide work in developing self-adapting applications. The challenges are:

1. how to handle underspecification of requested resources
2. how to handle overspecification of requested resources
3. time bounds on requests
4. how to measure current state quality
5. making accurate predictions
6. dealing with deceptive processes

In Figure 4, the sketch dictating what type of resources an application desires is a high-level request (`FAST + MULTIPLE + LENIENT + CORE`, line 2). What it means for a core to be fast is not clear. High-level sketches simplify the developer's task of describing which type of resources an application needs. High-level sketches may leave the Resource Decision Maker unable to reason about how two applications differ with regards to their resource requirements and usage. For example, on a multicore heterogeneous system running two multi-threaded applications, the first application may benefit from fast floating point operations, while the other application may benefit from fast memory access. If programmers do not provide hints about what it means for a core to be fast, then the Resource Decision Maker may be unable to appropriately assign cores.

With high-level sketches, the requested resources may be underspecified. A lack of sketch detail may be due to programmer error or limitations on sketch expressiveness. Online profiling may be useful in addressing this challenge.

Overspecification is also a concern. The developer may request a resource that may not easily be granted. For example, the developer may request exclusive access to all cores. Some cores may be in use by a higher priority user's application. Should the Resource Decision Maker cause the application to wait until the requested resources are available? What if it is unknown how much time will pass before the resources will become available? Some overspecified requests may be impossible to be met on a specific system (e.g., a request for a core with 128 KiB of L1 cache space). Should the system ignore such a request? What if the request is necessary for the application to function correctly? One way to address this challenge is to have the Resource Decision Maker lie about resources, while making a best effort to meet application requirements.

Time bounds on requests are another concern. Requests can be from the Resource Decision Maker to an application (e.g., to retrieve a resource), or from an application to the Resource Decision Maker (e.g., for an additional resource or change of resource). Quickly complying with requests allows both the Resource Decision Maker and applications to react quickly and reason more accurately about system behavior. Each component (applications and the Resource Decision Maker) should quickly comply to commands and queries. However, this may not always be possible. For example, if a thread is signaled to shutdown, it must do so at a convenient point. Terminating the thread immediately could put the application in an inconsistent state. If the application does not comply within a reasonable amount of time, what should the system do? Should the request or command be ignored? Should the application later be punished for not complying in a timely manner? The correct decision is unclear, and likely application, input, and system dependent.

Another challenge is how the Resource Decision Maker can accurately monitor system state and progress towards reaching its goals. What metrics should be used? Given a current configuration and resource assignment, is the system in a good state or a bad state? Should the system try to improve its configuration? Trying a new configuration may be costly (i.e., if the new configuration is inferior to the current one). Hardware performance counters or dynamic binary instrumentation may assist the Resource Decision Maker in gathering information to meet this challenge.

A fifth challenge is how the Resource Decision Maker can make accurate predictions. The Resource Decision Maker and applications have an incomplete understanding of system behavior. In Figure 4, the application explicitly shows that it can reduce cache contention. An ideal application would also be able to reason about how well it can reduce cache contention. For example, what will the average cost (in terms of cycles or joules, as appropriate) of a memory access decrease by if the application attempts to reduce cache contention? Such detailed reasoning requires knowledge of how cores access memory and may require information from across applications. Online learning and hardware performance counters may assist with this challenge.

The last challenge that we point out is that an application may be deceptive. For example, a multi-threaded program may request a minimum of four cores at all times. This may be a legitimate request (e.g., if the four cores allow the application to meet its QoS guarantees). However, the application may lie in order to gain more resources than it would normally be given, to let itself execute faster. Deceptive processes may be a result of a malicious programmer or programmer error.

These challenges apply to self-adaptive systems that use a request/grant model to obtain and release resources. To understand how to solve the challenges, we have developed and implemented rudimentary run-time support for inflate/deflate programs. With this rudimentary support, we have explored two scenarios to understand the burden on a programmer, whether the inflate/deflate model has promise, and what is needed from the Resource Decision Maker.

3. IMPLEMENTATION

We experimented with two simple scenarios where adaptation is beneficial: energy conservation and pipeline balancing. We modified two multi-threaded programs from the PARSEC benchmark suite[2] (blackscholes and dedup) and tested each in a sample scenario. Blackscholes calculates European options pricing using Black-Scholes partial differential equation. Dedup performs data stream compression using both global and local compression.

Blackscholes was tested in a sample scenario where a system power usage policy is in effect on a shared multicore system. The power usage policy, which changes throughout the day, dictates which resources (cores) are available at a given time. Additionally, this scenario required that blackscholes share CPU resources to ensure fairness and avoid starvation.

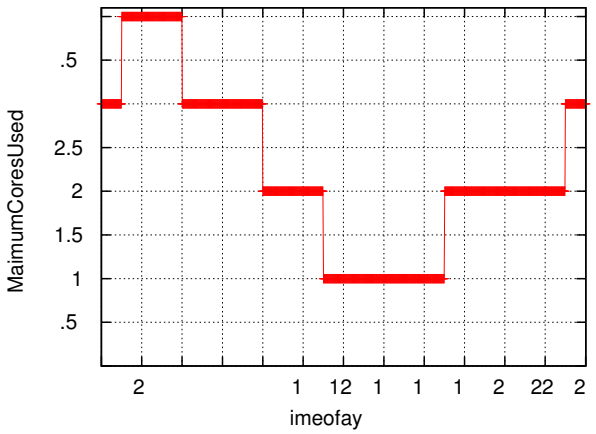


Figure 6: Sample core usage policy

Dedup was tested in a sample scenario where cores come online and go offline without warning. Dedup utilizes a multi-threaded pipeline programming model, wherein each thread performs the work of a pipeline stage. Threads therefore can be poorly allocated. For example, a thread may be created to perform work of a non-bottleneck stage. Our inflate/deflate-capable dedup, through proper dynamic resource allocation, avoids bottlenecks.

3.1 Energy Conservation Use Case

In this use case, we consider a scenario where a server is housed in a data center. The server continuously performs batch work and runs two programs: a multi-threaded application (blackscholes) and a single-threaded application which we refer to as the side process (described later). The goal of this exploration is to allow for applications to fairly and efficiently share cores in such a scenario.

The data center's power consumption is recorded by a digital smart meter. The smart meter periodically records power consumption to charge different dollar amounts per kilowatt hour as a function of the time of day, season, etc. The owners of the data center want to reduce server power usage when electricity prices are at their peak. However, servers cannot be turned off entirely during peak hours, or else no work would be accomplished.

In response to the data center owners' request, suppose the system administrator decided to institute a policy in which during certain hours, the maximum number of allowable active cores on an individual machine is limited. A reduction in the number of active cores can reduce overall power consumption. The Resource Decision Maker is used to implement this policy. To explore this use case, we modified blackscholes to adapt itself to core availability.

Figure 6 shows the administrators' sample policy for a four core machine. In the early afternoon, demand on local power plants is highest. Thus, the cost of electricity is highest during that time. To compensate for higher daytime prices, the policy dictates that fewer cores should be used. In the evening and morning times, when electricity costs less than the peak cost, more cores are used. Late at night, when electricity is cheapest, all cores on the system may be used.

If the only goal was to limit how many cores are active at any given time, then no application modification would be necessary. The system administrator could devise a system that uses Linux's `sched_setaffinity` to restrict which cores the processes and their threads are allowed to run on. Threads would automatically be restricted by the OS scheduler to run on a subset of the system's available cores.

Restricting the scheduling of process threads with `sched_setaffinity` has several disadvantages. To make full use of all the system's cores (i.e., as desired late at night) there must be as many threads as there are cores. Yet, when fewer cores are available (i.e., in the afternoon when electricity prices are highest), fewer threads are necessary. The standard blackscholes program can be spawned with a user-specifiable number of threads, but once specified, changes cannot occur at run-time. This limitation can lead to a mismatch between the number of CPU-bound threads and the number of usable CPU cores.

On a system with more cores than threads, cores will be idle and performance may suffer. A system with more threads than cores may cause threads to be starved for CPU time before they are scheduled again. More threads than cores can also unnecessarily increase the cost of context switching, due to threads' working sets not being kept in cache before the threads get rescheduled.

A more flexible system would feature adaptable processes. As the number of usable cores is decreased, multi-threaded applications should use fewer threads, until utilization is 100%. At a minimum, each application will be allowed to have one work thread active. As the number of usable cores is increased, multi-threaded applications should create new worker threads until the system is again at 100% utilization. This policy avoids oversubscription and unnecessarily high context switching overhead, yet ensures good performance.

We now describe how we modified blackscholes to implement this policy.

3.1.1 *Blackscholes Modification*

The base version of blackscholes accepts on the command line the desired number of worker threads. The workload is read from a file and stored in an array of length n. The desired number of threads is created. Each thread is given an identification number which is used to evenly divide the work among worker threads. Threads compute a result for each entry in the workload array. The main thread waits at a barrier for all worker threads to finish their assigned work. The main thread then writes the worker threads' results to a file and subsequently exits.

To support the dynamic addition and removal of worker threads, we modified blackscholes in the following way. We avoid static work allocation among threads by using a work unit list. The work unit list is a thread-safe linked list whose elements are tuples of the form $(start, end)$ where $0 <= start < end < n$. List initialization is done by the main thread. Each entry in the input work array is covered by a work unit list tuple's range. Worker threads execute a loop, popping a tuple from the work list on each iteration. After obtaining a tuple, a thread computes results for the elements in the array between the tuple's *start* and *end* indices.

If the work list is empty, then no more work is available and a thread quits. Before a new work unit is obtained, the thread checks if it has been sent a quit message. How quickly a program reacts to a message can be adjusted by changing the size (range) of work units. The size of work units is controlled by an environment variable. Larger work

System Property	Value
Processors	Intel® Xeon® E5335 @ 2.00GHz
number of cores	4
RAM	8 GiB
OS	Linux (kernel 2.6.29.6)

Figure 7: Test system properties

units cause messages to be checked less often, but may cause less lock contention in the work unit queue. Smaller work units cause quicker responses to thread messages but may incur work unit list locking overhead.

Other modifications to blackscholes include adding the code to connect to the Resource Decision Maker, a function to create new threads (in response to the Resource Decision Maker giving blackscholes another core resource), and a function to signal the shutdown of a worker thread (in response to the Resource Decision Maker requesting a core resource back). Overall modification difficulty was relatively easy.

We experimented with this policy and observed its affects on the throughput of blackscholes, as well as the policy's affect on other system processes. We compare with an unmodified blackscholes program.

3.1.2 *Experimental Results*

Experiments with the original core usage policy (Figure 6) would be 24 hours in length. To avoid such lengthy experiments, we use a shrunken policy. We break the experiment up into seven five-minute periods. The resulting experiment is 35 minutes in length. The first period allows only one core to be active. Each period thereafter allows one more core than previously allowed. In period four, four cores are allowed. This is the number of cores in our test system. Subsequent periods after the fourth period allow one fewer core than previous periods until, in period seven, only one core is allowed to be active. The test system runs blackscholes and an additional process, called the side process. Figure 7 shows the properties of our test system.

At the end of each of the seven periods in the experiment, we measure how many work units were computed by blackscholes and the side process. A blackscholes work unit is counted as completed when an array entry is computed. The side process performs calculations involving several memory and arithmetic operations (single-threaded, CPU bound). A side process work unit is one completed calculation. Work units for each program are normalized to the maximum number of work units completed by that program, in any period in either of the two experiments.

On our test system, blackscholes would finish its work before 35 minutes have elapsed. Periods smaller than five minutes might not allow the system to reach a stable state. To allow blackscholes to run throughout the length of the experiment, we modified blackscholes to process its input multiple times. We measure how many work units the unmodified and inflate/deflate-capable blackscholes completes in each period.

The side process is always given one core, except when the system is constrained to run entirely on one core, in which case the side process shares the core with blackscholes (as scheduled by the OS). We measure how many work units it completes when run alongside the inflate/deflate-capable blackscholes version. We also measure how many work units it completes when run alongside an unmodified blackscholes instance that has been allocated four threads.

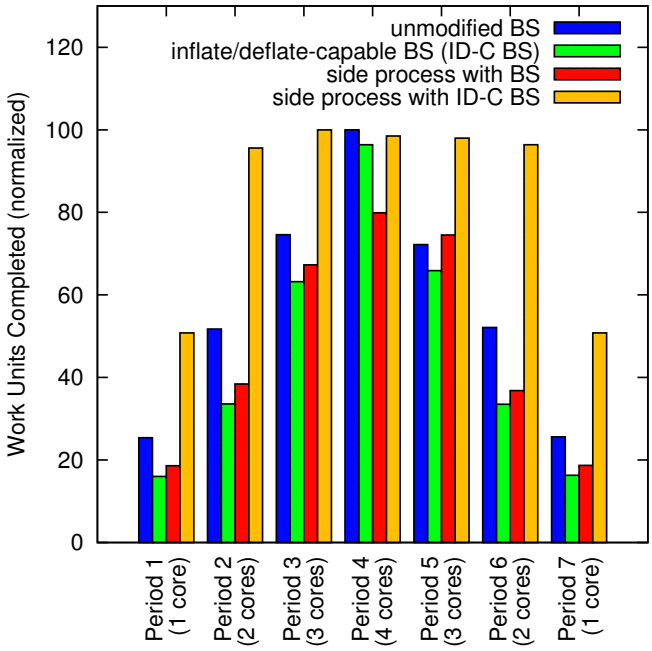


Figure 8: Blackscholes, side process work units completed

Figure 8 shows the results of this experiment. The height of the first two bars in each period shows the number of work units completed by the unmodified blackscholes (BS) and inflate/deflate-capable blackscholes (ID-C BS), respectively. In general, each version of blackscholes shows a similar pattern: as the number of allowed cores increases, so does the rate of work unit completion. Consistently, the unmodified version of blackscholes completes slightly more work units than the ID-C BS.

The reason that the unmodified blackscholes process consistently performs slightly better is the unmodified blackscholes process does not adapt how many threads it uses. When the system policy allows for few cores to be used, the unmodified blackscholes process still uses as many threads as if the system dictated that all cores were available. The unmodified blackscholes process, because of this fact, is given more CPU time by the OS. The OS may obtain additional CPU time by stealing it from the side process. Also, threads in the unmodified version of blackscholes do not make use of a shared work unit queue. This can avoid linked list contention.

Figure 8 also shows the relative number of work units completed by the side processes. These results are the third and fourth bars in each period. The third bar is the number of work units completed by the side process when the side process shares the system with the unmodified blackscholes instance. The fourth bar is the number of work units completed when the side process shares the system with the inflate/deflate-capable blackscholes process.

In general, the number of work units completed by the side process increases as the policy allows more cores to be used in each period. However, the version of blacksc-

Period	Efficiency (unmodified)	Efficiency (ID-C)
1	44.0	66.8
2	45.1	64.6
3	47.3	54.4
4	45.0	48.7
5	48.9	54.6
6	44.5	65.0
7	44.3	67.1

Figure 9: Energy efficiency

holes (unmodified, or inflate/deflate-capable) run with the side process heavily influences the number of work units the side process completes. When the number of cores available on the system is low (periods one, seven), the side process run with the ID-C BS process completes more than twice as many work units as the side process when run with the unmodified blackscholes process. When the system policy allows all cores to be used (period four) the side process running with ID-C BS completes significantly more work units than when the side process is run with an unmodified blackscholes process (23% more). In fact, when more than one core is available on the system (periods two through six) the side process running with the ID-C BS process always performs nearly at its maximum capability.

Figure 9 shows an efficiency comparison between the unmodified system and the inflate/deflate-capable system. Efficiency is defined as the number of normalized work units completed by both processes in a period, divided by the number of cores used in that period. The inflate/deflate-capable system (third column) is always more efficient than the unmodified system (second column), especially when core resources are few (e.g., the first and last period).

The use of the ID-C BS is clearly a winner in terms of fairness. The side process runs almost undisturbed because the Resource Decision Maker ensures that the ID-C BS process cannot steal CPU time. As the system policy allows for more or fewer cores to be used, the Resource Decision Maker adjusts ID-C BS appropriately. This ensures a fairer use of system resources, so that the side process does not suffer unnecessarily.

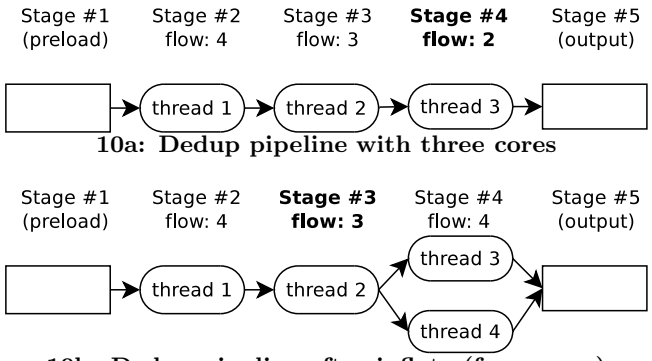
Without the Resource Decision Maker and ID-C BS, the blackscholes process must use a static number of threads throughout its lifetime. This static allocation risks the underutilization or overutilization of the system's cores. If the static allocation overutilizes the system's cores, the side process may suffer, as seen in Figure 8.

In conclusion, the inflate/deflate-capable blackscholes combined with the Resource Decision Maker, causes cores to be utilized more fairly and efficiently across processes. Fairer utilization allows the system to run with fewer active processors while still maintaining good performance, enabling power savings.

3.2 Balanced Pipeline Use Case

In this use case, we consider a multicore system in which the number of active cores that are available is unknown. Furthermore, the cores come online and go offline without warning[1]. The maximum number of cores available is also unknown in advance (e.g., cores may be hot pluggable).

[1]We assume processes are given adequate time to migrate their work from soon-to-be-offline cores.



10a: Dedup pipeline with three cores

10b: Dedup pipeline after inflate (four cores)

Figure 10: Dedup pipeline stages

A process running on such a system should maximize its usage of a newly available core. Similarly, if a core becomes unavailable, the application should appropriately respond to minimize the negative impact that a loss of a core may cause. To test such an environment we have modified dedup to be more flexible in order to fully take advantage of such a system.

Dedup utilizes a multi-stage pipeline. We modified dedup to support dynamic insertion and removal of threads in its pipeline stages. Furthermore, we instrument stages to calculate the flow of work units through stages. This allows the identification of slow stages, which can benefit from an additional thread running on a new core.

3.2.1 *Dedup Modification*

The base version of dedup is a five stage pipeline, as shown in Figure 10a. The program preloads an input file to reduce I/O during its parallel phase. This is the first stage's work. Next, three thread pools are created: one for each of the middle stages of the pipeline. The main thread waits for the middle stages to finish their work. After the middle stages are done, the last stage's work is performed (outputing the results to a file). The time when the middle three stages are doing their work is the region of interest (ROI). The third stage may bypass the fourth stage and send work units directly to the fifth stage.

A single command line argument (n) specifies how many threads are in each of the thread pools. With n threads per stage and three middle stages, $3 \cdot n$ threads are created. On a c core system which attempts to maximize throughput/performance, all c cores should have schedulable threads. To guarantee this constraint, $3 \cdot c$ threads must be spawned[2]. This is a limit of the static nature of dedup's work consumption.

Due to dedup's static thread creation, it is likely that the system can be oversubscribed. That is, with c cores and 3 stages, $3 \cdot c$ threads compete for CPU time on c cores. When a stage has no more work to perform, its threads will quit. Oversubscription therefore is only lessened as the program finishes its work.

Dedup's static thread creation also results in an inflexible use of CPU cores: once dedup's threads have been created, no more threads may be created. If new cores become available, dedup cannot create more threads to use the ad-

[2]In the worst case, only the fourth stage is unfinished.

ditional resources. Modifying dedup to support dynamic pipeline stage thread creation would help avoid oversubscription while also allowing for new resources to be used as they become available.

To enable dynamic pipeline adaptation we modified dedup to determine the slowest stage(s). This change required adding profiling code and counters. The throughput of each stage is found by measuring each stage's flow. The flow of a stage is computed as $flow(i) = \frac{threadCount_i}{difficulty_i}$, where $difficulty_i$ is a measure of the complexity of a stage, i. To approximate the complexity of a stage, we use average processor time spent (found via `clock()`) to complete a work unit in that stage ($\frac{\sum_{j=0}^{j=workUnits} TimeToFinishWorkUnit_j}{workUnits}$). As a result, stages which on average take longer to complete a work unit are considered more difficult.

$TimeToFinishWorkUnit_j$ is how much processor time worker threads consumed while processing a particular work unit. This time is added to a per-stage counter. Threads also increment a per-stage counter after completing a work unit ($numWorkUnits$). The application also maintain counters describing how many threads are allocated to each stage ($threadCount$). These counters are reset on demand (i.e., after adding or removing a thread from a stage) to recalculate the flow.

When the Resource Decision Maker offers a core to dedup, the stage with the smallest flow is given a thread. If two or more stages tie for lowest flow, the stage earlier in the pipeline is given a thread. The thread will run on the newly acquired core. An example of this behavior is shown in Figure 10a. In the figure, the fourth stage has the lowest flow. Dedup detects this and creates another thread to do the fourth stage's work. The result is the inflated version in Figure 10b. In the inflated version, the new bottleneck is the third stage. If another core was offered to dedup, the third stage would be granted another thread/core.

When a core is requested from dedup by the Resource Decision Maker, dedup finds the stage with the highest flow. A per-stage flag is then set. Threads check their stage's flag before they obtain their next work unit. If the flag is set, the thread exits and the flag is reset. Threads may also exit because no more work is available (i.e., previous stages have finished or run out of work). Once a stage's work is completely finished, any threads in that stage exit and the corresponding cores become idle. For each idle core, dedup finds the unfinished stage with the lowest flow and spawns a new thread on the core to accomplish the slow stage's work.

We next describe our experiments to observe the modified dedup's ability to dynamically adapt to resource availability.

3.2.2 Experimental Results

With the modified version of dedup, we wanted to experimentally determine two things. Firstly, we want to know whether the modified dedup performs work more efficiently than the unmodified version given the same resources. The modified dedup may be more efficient because it can reason about the flow of pipeline stages and does not oversubscribe cores. Secondly, we want to know how well the modified dedup responds to resources that dynamically come online.

Dedup's native input file is processed on our test system in approximately 30 seconds. To conduct longer experiments, we use a modifed 3.0 GiB test file. We conduct all experiments on the system described in Figure 7.

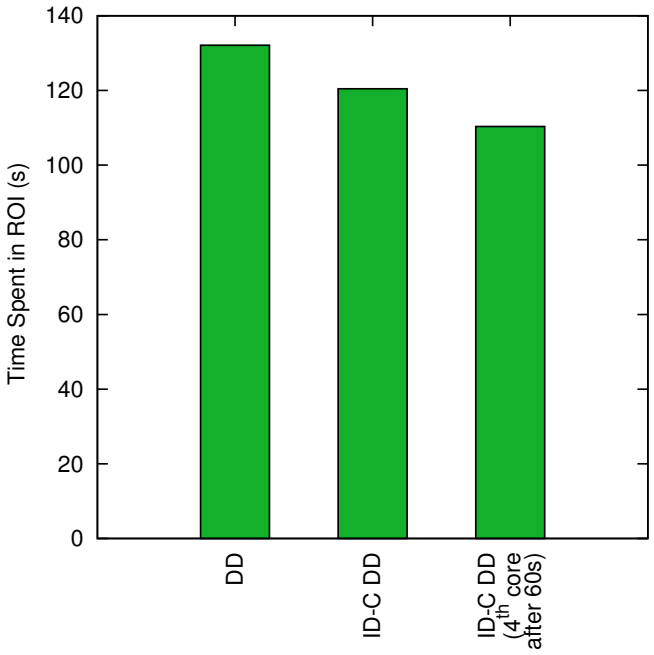


Figure 11: Dedup (DD) experimental results

To determine how much more quickly the modified dedup processes work versus the unmodified dedup, we separately run each version on our test system. Both versions are forced via Linux's `taskset` command to run on three cores. We measured how long dedup spends in its region of interest (ROI). We instruct the unmodified dedup to spawn three threads per pipeline stage (nine threads total).

The second experiment starts similarly to the first one. However, after 60 seconds in the ROI, we add a core (to four cores total). The core is given by the Resource Decision Maker to the modified dedup. The modified dedup creates a thread on it. The unmodified dedup is unable to respond to the availability of a new core; its performance is the same as in the first experiment.

Figure 11 shows the results of these experiments. The y-axis is the time spent in the ROI. Lower is better. The first bar shows the unmodified dedup's results (DD). The unmodfied dedup processes the input file in 132 seconds. The second bar shows the inflate-deflate capable dedup's ability to process the workload (ID-C DD). It processes the same input file in 120 seconds. The third bar shows the results of giving the modified dedup process an additional core after 60 seconds: the workload is processed in 110 seconds (a speedup of 17% over the unmodified version).

With the first two bars of Figure 11, both dedup versions are only given 3 cores yet the inflate-deflate capable dedup (ID-C DD) is 12 seconds faster. We attribute this to the use of fewer threads. With ID-C DD, each thread is always schedulable. No thread ever has to wait for a core to become available. Threads are unlikely to be context switched out so that another thread can run. This causes their working sets to remain across scheduling quantas. Additionally, with fewer threads, lock contention is minimized.

We also performed experiments where cores are dynamically taken from the inflate-deflate capable dedup process (not shown due to space constraints). The conclusions drawn

are similar to past conclusions: the modified dedup version is able to effectively manage its resource usage.

In conclusion, the modified version of dedup responds as available resources change. The source code modifications made were significantly more involved than the modifications made to blackscholes. However, dedup is also significantly more complicated than blackscholes. We expect these methods and results to be applicable to other pipeline-based multi-threaded programs.

4. RELATED WORK

There is much past work in the field of autonomic computing. Works like [12] provide valuable guidance (i.e., design patterns) for creating adaptive systems. Works like [6] provide ways of thinking about the capabilities of autonomic systems. Some previous work on autonomic computing ([1]) requires applications be built on top of a framework or platform from scratch. We are interested in exploring self-adaptive applications on self-adaptive systems, regardless of original application design or language.

Previous work on retrofitting legacy systems ([10][5][3]), focuses on large distributed systems, such as those in a grid. Here, the resource is an entire computer providing a service. If something happens to the computer, other computers may respond appropriately (e.g., by being turned on or off). We consider resources within a computer (e.g., cores and memory). We additionally are concerned with the interactions between processes on a system (e.g., cache contention) and how programs can be caused to interact and reach a better configuration than static policies allow.

[13] explores abstracting streaming programs for performance in order to adapt them at runtime to the host architecture. [8] abstracts machine resources and grants subsets of machine resources to applications to let applications meet individual performance goals (performance isolation). [9] allows for application-specific hypervisor policies. Our work is more general and focuses on letting applications properly respond to resources appearing, disappearing, or changing.

Programming language extensions to enable better performance have also been proposed. X10[4] and Cilk++[7] enable better and easier system utilization through extending existing languages (Java and C++ respectively). X10 is intended for non-uniform cluster environments, while Cilk++ is for individual multicore computers. Our inflate/deflate paradigm focuses on dynamically changing resources.

5. CONCLUSION

In this paper we have presented a new programming model which allows for dynamic program adaptation and resource management through the concepts of inflation and deflation. To explore the difficulty and rewards of creating programs which are inflate/deflate-capable, we modified two PARSEC applications (blackscholes and dedup). Each application was then tested in its own use case scenario.

Blackscholes was tested in a scenario where a power usage policy was in effect on a system with multiple workloads. The inflate/deflate-capable blackscholes was able to achieve good throughput without sacrificing the performance of another system process, resulting in overall increases in efficiency. We then tested an inflate/deflate-capable version of dedup in a scenario where compute cores dynamically arrive and leave. Dedup completed its workload faster than the unmodified version and was additionally able to dynamically adapt itself to core availability at runtime. Modification of blackscholes and dedup to support inflate/deflate was non-trivial but worthwhile. Based on our modifications and experiments, we believe that the inflate/deflate paradigm is a viable way to create autonomic programs and systems.

6. ACKNOWLEDGEMENTS

This work was supported in part by NSF awards CNS-1012070, CCF-0811295, CCF-0811352, and CNS-0702236.

REFERENCES

[1] M. Agarwal, V. Bhat, H. Liu, V. Matossian, V. Putty, C. Schmidt, G. Zhang, L. Zhen, M. Parashar, B. Khargharia, and S. Hariri. Automate: enabling autonomic applications on the grid. In *Autonomic Computing Workshop*, June 2003.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *PACT '08*. ACM, 2008.

[3] S. Chandra, X. Li, T. Saif, and M. Parashar. Enabling scalable parallel implementations of structured adaptive mesh refinement applications. *J. Supercomput.*, 39, February 2007.

[4] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. OOPSLA '05. ACM, 2005.

[5] S. Diwan and D. Gannon. Adaptive resource utilization and remote access capabilities in high-performance distributed systems: The open hpc++ approach. *Cluster Computing*, 3:1–14, January 2000.

[6] S. Dustdar, C. Dorn, F. Li, L. Baresi, G. Cabri, C. Pautasso, and F. Zambonelli. A roadmap towards sustainable self-aware service systems. In *SEAMS '10*. ACM.

[7] C. E. Leiserson. The cilk++ concurrency platform. DAC '09. ACM, 2009.

[8] K. J. Nesbit. *Virtual private machines: a resource abstraction for multicore computer systems*. PhD thesis, Madison, WI, USA, 2009.

[9] D. Nikolopoulos, G. Back, J. Tripathi, and M. Curtis-Maury. Vt-asos: Holistic system software customization for many cores. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE Int'l Symp. on*, 2008.

[10] J. Parekh, G. Kaiser, P. Gross, and G. Valetto. Retrofitting autonomic capabilities onto legacy systems. *Cluster Computing*, 9, 2006.

[11] C. Pheatt. Intel®threading building blocks. *J. Comput. Small Coll.*, 23, April 2008.

[12] A. J. Ramirez and B. H. C. Cheng. Design patterns for developing dynamically adaptive systems. In *SEAMS '10*. ACM, 2010.

[13] D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe. A lightweight streaming layer for multicore execution. *SIGARCH Comput. Archit. News*, 36, May 2008.