A Platform for Developing Adaptable Multicore Applications

Dan Fay  
Department of Electrical, Computer, and Energy Engineering  
University of Colorado  
Boulder, CO 80309-0430  
faydr@colorado.edu

Li Shang  
Department of Electrical, Computer, and Energy Engineering  
University of Colorado  
Boulder, CO 80309-0430  
li.shang@colorado.edu

Dirk Grunwald  
Department of Computer Science  
University of Colorado  
Boulder, CO 80309-0430  
grunwald@colorado.edu

ABSTRACT

Computer systems are resource constrained. Application adaptation is a useful way to optimize system resource usage while satisfying the application performance constraints. Previous adaptation efforts, however, were ad-hoc, time-consuming, and highly application-specific with limited portability between computer systems. In this work, our goal is to provide a development platform to systematically explore and rigorously apply portable application-specific runtime optimization. We present OCCAM, a software platform for developing multicore adaptive applications. OCCAM’s design-time platform consists of APIs and data structures that allow application developers to specify the performance constraints and application-specific optimization techniques. OCCAM’s run-time system dynamically manages the application behavior and optimizes system resource usage. OCCAM targets emerging Recognition, Mining, and Synthesis Applications (RMS). Using a set of RMS benchmarks, the experimental study demonstrates that OCCAM can successfully optimize resource usage under application performance constraints across a wide range of computer platforms, with an average of 38% energy savings on an Intel Atom-based, energy-constrained portable system, and an average of 24% energy savings on a high-performance, dual-core computer platform. These savings are accomplished with low overhead. We have also successfully extended OCCAM applications to run on a 16-core setup.

Categories and Subject Descriptors
D.1.3 [Software]: Programming Techniques—Parallel Programming

General Terms
Design, Management, Performance

Keywords
Multicore, Application Adaptation, Run-Time Systems, Parallelization, Frequency Scaling

1. INTRODUCTION

Computer system performance optimization is resource constrained. In early computer systems, programs were limited by small memory capacities and computation capabilities; today, programs are constrained by tight energy budgets in mobile systems or by stringent thermal constraints in high-performance computer systems. Application adaptation, which leverages application-specific knowledge about the program to dynamically adjust the program behavior and resource usage, is an effective way to optimize system resource usage while satisfying the application’s performance requirements. Examples of application adaptation include varying the resolution of an input image in order to trade result quality for computation complexity; changing the blocking size of an algorithm to improve the cache hit rate; and reducing an application’s throughput to reduce CPU temperature.

Existing research in application adaptation [3, 14, 15, 19, 22] lacks systematic design-time and run-time support. Application optimization was mostly done manually using domain-specific knowledge, which is ad hoc and time-consuming. Each adaptable application is a one-off design that lacks a common run-time system to systematically direct the application’s online adaptation. Since developers only have limited knowledge of system resource availability and application behavior at run time, system performance and resource usage are suboptimal. Furthermore, it is impossible for applications to accommodate new performance requirements or system resource constraints without rewriting the application. The move from single core to multicore systems further complicates the design of adaptable applications. There is an increasing need for design-time and run-time support for developing adaptable multicore applications.

Existing general-purpose run-times are not suitable for this job [5] [4]. They lack the capability to automatically acquire and comprehend application-specific knowledge, such as application performance requirements and domain-specific control and optimization techniques. Without such knowledge, a run-time system can only adapt applications in limited ways. Developers, on the other hand, are able to identify and provide application-specific knowledge, such as ap-
application throughput and result quality requirements, as well as specific control and optimization techniques to trade off between computation–communication–memory resource usage and result quality. However, existing run-time systems do not have the capability to systematically interact with developers and rigorously leverage application-specific knowledge.

To tackle these problems, we propose and develop a development platform to systematically explore and rigorously apply portable, application-specific run-time optimization. Our approach, called the OCCAM (Optimizing Constrained Concurrent Applications at runtIME) platform, is a middleware framework that provides APIs and data structures for specifying the program performance requirements and how to adapt the application, along with a run-time system that adapts applications at run time to optimize system resource usage under the application performance constraints. We designed the OCCAM platform to target emerging Recognition, Mining, and Synthesis workloads like the ones described in [7]. The OCCAM platform is comprised of several components: the OCCAM API, which the programmer uses to specify the performance requirements and how the application trades off result quality for computation complexity; the Data Pyramid, which the application uses to access data; and the OCCAM runtime system, which optimizes system resource usage while satisfying application performance constraints.

OCCAM makes the following contributions:

1. A design-time platform for application adaptation. This platform allows a developer to specify an application’s performance requirements and its application-specific adaptation techniques for optimization by the run-time system.
2. A control-based run-time system for optimizing system resource usage of a variety of applications within the application’s performance constraints.
3. Enables portability between different computer systems by allowing new resource constraints to be added to the run-time system without having to rewrite or recompile the application.
4. A framework for optimizing applications running on multicore systems.

The rest of this paper is structured as follows. Section 2 describes the OCCAM platform. Section 3 describes the test platform, benchmarks, and experimental results. Section 4 describes related work. Section 5 summarizes and concludes the work.

2. THE OCCAM PLATFORM

As shown in Figure 1, the OCCAM platform consists of three main components: the OCCAM API, the Data Pyramid, and the OCCAM run-time system. The OCCAM API provides the interface used by application developers to systematically specify application performance requirements and application-specific control and optimization techniques. The Data Pyramid is a data structure that provides input data to the application and accepts computed result data from the application. The OCCAM run-time system optimizes computer system resource usage, e.g., power consumption, while satisfying application performance requirements.

Using the OCCAM API, an application designer specifies two types of performance constraints: result quality constraints and real-time throughput constraints. OCCAM requires the designer to provide a function to evaluate whether the result meets the performance constraints, but only requires a time value to be provided to evaluate the system’s real-time throughput. The OCCAM API also provides a structured way to relax the application’s performance constraints by allowing the application designer to specify a series of progressively less-desirable constraints. Providing structured constraint relaxation allows OCCAM to optimize an application on computer systems where limited computation resources make it impossible to meet the application’s baseline constraints.

The OCCAM run-time system is a control-based system. At run time, it rigorously adapts application behavior and optimizes system resource usage to conform to application performance constraints. Besides application-specific knowledge, OCCAM also supports general-purpose run-time adaptation and optimization techniques. General-purpose techniques include both software-based methods, e.g., multi-core scheduling; and hardware-based mechanisms, e.g., dynamic voltage and frequency scaling (DVFS) and power- and thermal-aware throttling. Application-specific and general-purpose techniques are leveraged in unison for run-time application and system adaptation, yet managed separately to maximize the reusability of the runtime. Only application-specific techniques need to be specified and incorporated in order for a run-time system to support new applications.

2.1 Computational Model

OCCAM uses a model of computation based on the stream programming model. Stream programming is based on two components: a stream, and a kernel. A stream is an array of elements that can be operated on in parallel, while the kernel performs work on each of the elements in the stream. The parallelism provided by this programming model provides an efficient way to partition and schedule an application for parallel execution on multiple processing cores.

This programming model has been used successfully in the past by stream computing languages such as Brook [16] and StreamIT [13] to describe a wide variety of algorithms for execution on parallel systems with varying numbers of compute resources.

Moreover, stream-based computing has been successfully used to provide scalable parallelism on a variety of other hardware systems besides multicore microprocessors, such as GPUs [6] and the Cell microprocessor [24]. By leveraging the streaming programming model, OCCAM can potentially support these important future architectures in addition to the multicore systems studied in this paper.

OCCAM modifies this stream computational model in several ways:

1. Streams are grouped into units called Throughput Tasks.
2. A new Throughput Task cannot be started until after the previous Throughput Task has completed.
3. How well the system is meeting its performance constraints is assessed after each Throughput Task.
4. The application can only be adapted between Throughput Tasks.
Dividing an OCCAM application into a series of parallel, coarse-grained, throughput-oriented tasks provides several advantages. First, many RMS applications map naturally to this task model. For example, in a sequence-of-frames application such as 3D modelling, each frame can be mapped to a Throughput Task. Similarly, for a data mining application, collections of data can be mapped to separate tasks. Second, this sequential task model facilitates effective, low-overhead control of OCCAM applications by the run-time. It provides a non-application-specific way to measure the application's quality and throughput. Moreover, the boundaries between tasks provide coarse-grained points to evaluate and apply control decisions. Finally, this model maps well to execution on auxiliary computational units such as GPUs. A task can be sent to an auxiliary processing unit, executed, and sent back without having to deal with complicated synchronization.

### 2.2 OCCAM API and the Data Pyramid

Applications interface with OCCAM via the OCCAM API and the Data Pyramid. The OCCAM API provides an interface for specifying an application's performance constraints, and for specifying how to adapt the application. The Data Pyramid provides an interface between the application and the OCCAM run-time system for the input data and the output data.

An application designer specifies two performance constraints via the OCCAM API: a result quality constraint and a real-time throughput constraint. OCCAM requires the designer to provide a function to evaluate whether the result produced at the end of a Throughput Task meets the performance constraints. While general techniques exist, such as interval arithmetic, to quantify the numeric error of a result, they cannot determine what the application deems correct. As a result, the designer must provide a way for OCCAM to determine whether a result meets the performance constraints. Due to OCCAM's computational model, it is much simpler to specify the real-time throughput requirements: all that needs to be provided is the maximum amount of time allowed for the execution of a single Throughput Task.

The OCCAM API also provides a structured way to relax an application’s performance constraints by allowing the application designer to specify a series of progressively less-desirable constraints. Providing structured constraint relaxation allows OCCAM to optimize an application on computer systems where limited computation resources make it impossible to meet the application’s baseline constraints.

OCCAM leverages the stream programming model to allow an application to run on a variable number of processor cores as well as to trade off result quality for computation complexity. Each data point within a stream can be processed independently of the other data points, exposing large amounts of parallelism for the OCCAM run-time system. Using this programming model, trading off result quality for computation complexity involves processing fewer data elements per Throughput Task by reducing the data resolution. Reducing the data resolution entails representing the information contained in the Throughput Task’s streams with fewer data points that provide a lower quality representation of the information contained within the Throughput Task.

The data resolution concept is useful for several reasons. First, many important classes of algorithms can be easily implemented using this design pattern. Many Recognition, Mining, and Synthesis applications, which are an important class of emerging applications, map well to this design pattern. Moreover, many multimedia, DSP, and image processing algorithms can utilize this concept with little modification: video and image processing algorithms can simply change the resolution of the images being processed, while many DSP algorithms can lower their sampling rate. Similarly, approximate matrix multiplication, as described in [23], can scale down its data resolution by reducing the number of matrix rows and columns computed. Search algorithms can likewise lower their data resolutions by processing fewer records. Finally, [8] showed that many machine learning algorithms can be expressed as a sum over points. Scaling down the data resolution with these algorithms should be possible by reducing the number of data points processed and correspondingly scaling up the result.

OCCAM’s data resolution adaptation is important for multicore applications because it is orthogonal to the parallel adaptability of an application. Scaling up or down the data resolution only changes the number of data points that need to be processed: the remaining data points can still be partitioned and scheduled to run on multiple cores. Scaling down the data resolution also scales down the working set size and the amount of memory bandwidth used by the application.
reduce communication overhead and cache misses. Scaling down memory bandwidth is important because memory bandwidth is becoming a major limiting factor within multicore microprocessors for many applications [18].

Applications interface with OCCAM’s parallel application adaptation framework via a special data structure called the Data Pyramid. The Data Pyramid is conceptually similar to an image pyramid in image processing; like an image pyramid, it can provide a Throughput Task’s stream data at various resolutions. The Data Pyramid is implemented as an inheritable C++ class. Implementing the Data Pyramid as a C++ class allows a designer to customize the Data Pyramid for an application. To facilitate moving data between the application and the Data Pyramid, OCCAM provides a dataObject class as a generic container. By inheriting from this class, an application can specify what kind of data should be carried within the dataObject. The Data Pyramid provides the following interface to the application:

1. **getData()**. An application uses the getData() method to obtain the input data for the current Throughput Task. This method must be provided by the application designer, since Data Resolution scaling is highly application-specific. getData() takes no arguments, and returns a pointer to a dataObject.

2. **putData()**. An application uses putData() to output the finished data for a Throughput Task. The purpose of putData() is to perform any application-specific postprocessing of the output data, such as upscaling or filtering. Like getData(), putData() must be provided by the application designer because processing the output data is highly application-specific. Calling this method also signals to the OCCAM run-time system that the application has finished processing a Throughput Task. putData() takes a dataObject reference as an argument, and returns nothing.

### 2.3 OCCAM Run-Time System

The OCCAM run-time system is a control-based system that optimizes the system based on its past behavior. It optimizes the system resource usage by adapting the application and computer system while ensuring that the performance constraints of the system are met.

OCCAM optimizes the system within the constraints of the application and the computer system by concurrently adapting both the application and the computer system. Partitioning the performance constraints and adaptation techniques into application and computer system groups allows the application to be portable among different computer systems. Optimizing the application on a computer system with new constraints and/or new adaptation parameters only involves modifying the run-time system to support them. Examples of computer system constraints include limited processor resources, energy consumption constraints, or thermal constraints. Examples of ways to adapt the computer system include scheduling work across different numbers of available CPU cores, using the CPUs’ DVFS to trade off lower performance for reduced power consumption, and throttling overheating CPU cores.

The OCCAM run-time system leverages efficient control techniques to provide stability, fast response, and a quick settling time. Error is minimized by meeting the performance requirement with minimal resource usage. With OCCAM, a positive error indicates that one or more of the system’s constraints are not being met. Positive errors occur when, for example, the result quality is not high enough or the system is not meeting its real-time throughput constraints because the CPU cores are running at too low of a frequency. A negative error, on the other hand, indicates that the system is not performing at its maximum efficiency due to slack in the system. A negative error, for example, can occur when the result quality is higher than needed or the system meets its real-time throughput requirements faster than needed because the CPU cores are running at too high of a frequency. Such data redundancy and system slack should be minimized in order to optimize the system’s resources.

The OCCAM run-time system is designed to adapt the system in discrete steps, rather than continuously. This design pattern better resembles the way real applications work. For example, application data resolution options such as image resolutions or DSP sampling rates map to a set of discrete options rather than to a continuous spectrum of choices. Likewise, computer systems have discrete numbers of processing cores and DVFS options.

As shown in Figure 1, the OCCAM run-time system is implemented as a hierarchy of three controllers: the Application Controller, the Computer System Controller, and the Contingency Controller. The three controllers optimize the system sequentially. That is, the Application Controller first optimizes the Data Resolution, then the Computer System Controller optimizes the computer system, and finally the Contingency Controller optimizes the constraints.

The PID controller was chosen for OCCAM’s controllers because it is a simple, effective feedback controller that demonstrates the adaptation capabilities of OCCAM. We are currently investigating using stochastic optimal control techniques to provide optimal or near-optimal control of OCCAM applications.

The Application Controller adapts the data resolution for each Throughput Task so that the result quality and performance requirements of the application are just barely met, which optimizes the system by removing redundant data elements. The Computer System Controller is responsible for optimizing the computer system so that the application’s real-time throughput constraints as well as the computer system’s constraints are efficiently met. In this work, the System Controller optimizes the application’s multicore characteristics using Intel’s Threading Building Blocks’ (TBB) [9] runtime to partition and schedule the workload across the computer system’s multiple cores. TBB adapts to application heterogeneity by load balancing using a work-stealing algorithm. The Computer System Controller also adapts the computer system using the processor’s DVFS features. The controller performs DVFS adaptation using a PID controller that keeps the CPUs’ frequencies and voltages as low as possible while still meeting the application’s real-time constraints. If the system cannot meet its performance constraints, the Contingency Controller provides a structured way to relax the performance constraints.

### 2.4 Writing and Executing an OCCAM Application

Writing an application for OCCAM consists of the following steps. First, the developer writes the application in a data-parallel form. Second, the developer provides
a `getQuality()` function that describes the quality of the output data. Third, the developer provides `getData()` and `putData()` functions that respectively handle scaling down and scaling up the Data Resolution of the input and output data. Finally, the developer provides a series of application-level constraints that allow OCCAM to relax and tighten the application’s constraints as needed.

The execution of OCCAM consists of sequentially executing a series of parallel, throughput-oriented tasks. During each task, OCCAM and the application perform the following steps, as shown in Algorithm 1. First, the application requests input data and parameters from the OCCAM run-time system. Next, the application prepares one or more streams to be sent to the OCCAM run-time system for execution. Next, the OCCAM run-time system partitions and schedules the stream(s) into subtasks for parallel execution on a multicore computer system. Once the execution is complete, the OCCAM run-time system evaluates whether the performance constraints of the application and computer system were met. Finally, the OCCAM run-time system adapts the application and computer system based on how well the performance constraints were met.

Algorithm 1 Execution Process of an OCCAM Application

```
T := Whether another task exists
while T == true do
  ts := getTime()
  data := getData()
  subtasks := split(data)
  for all subtasks do
    Schedule subtask for execution
  end for
  for all subtasks do
    result := fold(subtask, result)
  end for
  putData(result)
  tf := getTime()
  time := tf - ts
  q := getQuality()
  p := getPower()
  controlQuality(q)
  controlDVFS(time, power)
  controlConstraints()
end while
```

3. EXPERIMENTAL RESULTS

This section evaluates OCCAM, the proposed development and run-time platform for developing adaptable multicore applications. OCCAM is designed to facilitate adapting emerging Recognition, Mining, and Synthesis (RMS) [7] applications. We developed seven RMS benchmark applications using OCCAM: three recognition benchmarks (`histogram`, `lr`, and `stereo_vision`), one mining benchmark (`wordcount`), and three synthesis benchmarks (`tachyon`, `seismic`, and `hearing_aid`). OCCAM can successfully optimize system resource usage under application performance constraints across a wide range of computer platforms. Three test platforms are considered: a low-performance, energy-constrained Intel Atom-based single-core mobile platform (referred to as LO); a high-performance, dual-core platform (referred to as HI), and a 16-core server system (referred to as MULTI). Each benchmark is developed using OCCAM, compiled once, and executed on both these platforms with the support of OCCAM’s runtime.

OCCAM’s performance is compared against a baseline configuration on each hardware platform using the popular Linux DVFS daemon `powernowd` [12] to control the processor performance and power efficiency. Overall, OCCAM improves the power efficiency of the HI system by an average of 24% and the LO system by an average of 38% versus the baseline single-core, `powernowd`-controlled configuration. OCCAM provides these power savings while imposing a maximum of a 5.79% overhead for the seismic benchmark. OCCAM’s better performance stems from its knowledge of the application. Unlike `powernowd`, which bases its DVFS control decisions solely on CPU utilization, OCCAM can make DVFS decisions based on the applications’ performance requirements. Moreover, in the event that the computer system lacks sufficient compute resources to meet the performance constraints, OCCAM’s Contingency Controller ensures good, continued control of the system by relaxing the application’s constraints.

3.1 RMS Benchmark Applications

OCCAM is designed to facilitate making emerging RMS applications [7] adaptable. We focus on RMS applications for several reasons. First, RMS applications will become increasingly important in the future as growing amounts of data increasingly require sophisticated applications to make sense of it. Second, RMS applications are highly computation–communication–storage intensive, leaving them heavily resource-constrained on existing and future computer systems. Moreover, RMS applications are highly parallel in nature, which means that they will be able to take full advantage of current and future parallel processing devices (multicore, GPGPU, etc.). Finally, RMS applications are an excellent candidate for application adaptation, as they inherently contain error tolerance, which makes it feasible for them to trade off result quality for computational complexity.

We implemented and/or modified seven RMS benchmark applications using OCCAM: three recognition benchmarks (`histogram`, `lr`, and `stereo_vision`), one mining benchmark (`wordcount`), and three synthesis benchmarks (`tachyon`, `seismic`, and `hearing_aid`). We plan to release these benchmarks to help other researchers study parallel, adaptable RMS applications.

`histogram` is derived from the like-named benchmark used in the Phoenix [21] system, an implementation of Google’s MapReduce [10] for multi-core systems. `histogram` represents a basic machine learning classifier that provides a statistical distribution. `histogram` trades result quality for computational complexity by evaluating fewer data points and then scaling up the output to provide an approximate result. `histogram` provides to OCCAM a result quality function that estimates the result quality using the sampling theorem.

`lr` is a recognition benchmark which also comes from the Phoenix benchmark suite. `lr` represents a widely-used machine learning classifier: linear regression analysis. Like `histogram`, the OCCAM version of `lr` trades result quality for computational complexity by processing fewer data points. `lr` estimates result quality using the sampling theorem.
<table>
<thead>
<tr>
<th>System</th>
<th>Processor</th>
<th>Frequency</th>
<th>L2 Cache</th>
<th>Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>LO</td>
<td>Atom</td>
<td>1.6 GHz</td>
<td>0.5 MB</td>
<td>2</td>
</tr>
<tr>
<td>HI</td>
<td>Core 2 Duo</td>
<td>2.0 GHz</td>
<td>4 MB</td>
<td>2</td>
</tr>
<tr>
<td>MULTI</td>
<td>Xeon X7350</td>
<td>2.93 GHz</td>
<td>4 MB</td>
<td>16</td>
</tr>
</tbody>
</table>

† Logical contexts (SMT); ‡ Shared between two cores.

Table 1: Hardware Configurations tested.

**Stereo vision** is a recognition benchmark derived from an NVIDIA CUDA application [2]. **Stereo Vision** computes the stereo vision distance for a pair of images using a parallel block matching algorithm. It trades off the ability to see far away objects for reduced computational complexity by reducing the horizontal resolution of the image. **Stereo vision** measures the resulting quality by counting the number of pixels in the result image whose distance cannot be resolved. The input stereo images come from [1].

**Wordcount** is a mining benchmark derived from the like-named benchmark in Phoenix. **Wordcount** parses a text file and tracks the number of occurrences of words in the text file. It is parallelized by dividing up the text file into smaller pieces that are then independently processed. **Wordcount** trades off computational complexity for result quality by only processing parts of the text file and scaling up the resulting count. It estimates the result quality using the sampling theorem.

**Tachyon** is a synthesis benchmark based on a parallel ray-tracer application shipped with TBB, which is based off of the tachyon benchmark in ALPBench [17]. **Tachyon** is parallelized by allowing each horizontal line in the output image to be rendered in parallel. Trading result quality for computational complexity occurs by changing the resolution of the outputted image. **Tachyon’s** result quality is a direct function of the current data resolution.

**Seismic** is a synthesis benchmark which visually simulates seismic waves as they propagate through water and different types of bedrock. **Seismic** is parallelized by dividing up the image to be rendered into tiles which can then be concurrently executed. Versus the original TBB example, the OCCAM version of **Seismic** renders only tiles with seismic waves above a certain intensity threshold. Raising this threshold allows **Seismic** to trade result quality for reduced computational complexity by reducing the number of tiles that need to be rendered in a Throughput Task.

**Hearing Aid** is a synthesis benchmark which is based on the sampling rate-adaptive hearing aid algorithm proposed by William Dieter in [11]. **Hearing Aid** simulates a hearing aid application that can reduce its computational complexity by reducing the sampling rate of the sound signal.

**OCCAM**’s overall ability to optimize system resource usage is measured using the system’s energy consumption required to complete each benchmark run. These measurements are used to show two things: how well OCCAM adapts applications for multicore systems (application-specific adaptation), and how well OCCAM can adapt the computer system’s DVFS settings (general-purpose adaptation). These energy consumption results are normalized to be relative to the baseline single-core setup where **powernowd** controls the computer system’s DVFS settings. **Powernowd** is a popular, open source DVFS daemon widely used in various Linux distributions. **Powernowd** controls the system’s DVFS decisions based on the CPU utilization. **Powernowd** uses an heuristic where if the CPU utilization goes above 80%, **powernowd** raises the CPU frequency to its maximum. If the CPU utilization falls below 20%, then **powernowd** lowers the CPU frequency by one DVFS step.

Eight test settings were performed by permuting the following three configurations – (1) the hardware platform tested (the high performance system (HI) versus the low performance system (LO)); (2) the method of DVFS control used (either OCCAM or powernowd); and (3) whether multicore adaptation was used (sp – no multicore adaptation used versus mp – multicore adaptation used).

Detailed studies are made of OCCAM’s ability to control the application and computer system at run time by capturing, at every Throughput Task, the instantaneous power consumption, CPU frequencies, and real-time throughput information. This system information is obtained using various microarchitectural interfaces. Power consumption is obtained using Linux’s ACPI drivers, the CPU frequencies are obtained using Linux’s CPU frequency scaling interface (provided through Linux’s /sys interface), and the real-time throughput is measured by using system timers to measure the amount of time the tasks take to execute. OCCAM’s and **powernowd**’s DVFS control are compared over time by plotting the instantaneous power consumption and the CPUs’ frequencies as the benchmarks execute. OCCAM’s overhead is measured using the gprof profiling tool. Overhead is measured by the percentage of execution time spent in OCCAM-specific functions and methods, such as **getdata()** and **getquality()**.

Finally, OCCAM’s ability to meet the application performance constraints was studied. The first comparison compares how well the applications’ real-time throughput constraints are met using OCCAM to adapt the computer system’s DVFS settings versus using **powernowd** to do so. The next comparison assesses the efficacy of the Contingency Controller by evaluating how well OCCAM can meet the application’s real-time throughput constraints with and without the Contingency Controller active. This comparison is done with the resource-limited, low performance (LO) system.

### 3.2 Test Platform

The three test platforms include a low-performance, energy-constrained, 1.6 GHz Intel Atom-based ASUS Eee PC netbook (referred to as LO), a high-performance, 2.0 GHz Intel Dual-Core computer platform (referred to as HI), and a 16-core, 2.93 GHz Intel X7300 system (referred to as MULTI). The Core 2 Duo platform contains two cores whose frequencies must be scaled together, while the Atom processor is a single core with two logical contexts (e.g., it supports Simultaneous Multithreading (SMT)). System power consumption was measured by reading the power drain using ACPI. Power consumption readings were taken at the end of each task.
Figure 2: Energy consumption comparison.

Figure 3: OCCAM vs. powernowd: HI's power consumption and CPU frequency over time.
Figure 4: OCCAM vs. powernowd: success at controlling the application over time.

only a single core microprocessor, it benefits from multicore adaptation due to its SMT support. For the energy consumption experiments, the Contingency Controller was disabled, and as a result, LO does not meet the application’s real-time throughput constraints. Its energy consumption improves because the higher performance of the multicore version allows the benchmarks to finish faster. In addition, by leveraging application-specific knowledge, OCCAM offers more power- and energy-efficient DVFS control. Since powernowd bases its DVFS decisions on CPU utilization, when a Throughput Task starts, powernowd raises the CPU frequency to the maximum. OCCAM, on the other hand, knows how much time the Throughput Task has to execute and keeps the CPU frequency just high enough to complete the Throughput Task.

Figure 3 shows the HI system’s power consumption and the DVFS settings across time for both OCCAM and powernowd. For all of the benchmarks except stereo_vision, OCCAM keeps the CPU frequency better controlled. While OCCAM keeps the CPU frequency mostly within a relatively tight band of one or two intermediate DVFS steps, powernowd widely fluctuates between the highest (2.0GHz) and lowest (0.8GHz) frequencies available on HI. While not shown in Figure 3 due to space constraints, LO also benefits from OCCAM’s better DVFS control because OCCAM keeps the CPUs consistently at the processor’s maximum frequency, allowing the benchmark to finish faster than it does with powernowd. Preliminary results with stereo_vision on MULTI, as shown in Figure 5, shows similar results: due to OCCAM’s application-specific knowledge, OCCAM is able to keep the CPUs’ frequencies at a much lower average level than can powernowd.

Figure 4 shows OCCAM’s versus powernowd’s ability to control the system to meet the application’s real-time constraints. For all of the applications except seismic, OCCAM achieves a lower error than powernowd. A lower error means that OCCAM better controls the applications to meet the application performance constraints while minimizing the system’s resource usage. OCCAM controls these applications better because it knows what the applications’ performance requirements are, and can make better CPU resource decisions based on this information. powernowd controls seismic better than OCCAM does because the non-linear adaptability characteristics of seismic make it difficult to control with OCCAM’s PID controllers.

OCCAM also responds to changes in the system faster than does powernowd. OCCAM shows this benefit most in histogram and lr. By 0.5s into histogram’s execution, OCCAM has reduced the error to −0.04, while at 5.03s, powernowd’s error is at 4.89. Likewise, OCCAM gets lr’s error down to −0.012 in 1.14s, while powernowd still has an error of 2.39 at 2.57s.

OCCAM’s ability to relax the application’s constraints allows for better control of the applications and the computer system on the LO platform in Figure 4. The Fidelity Index value indicates at what level OCCAM has relaxed the constraints: the value of 0 indicates the baseline constraint, while higher values indicate progressively relaxed constraints. For all of the benchmarks, the LO platform does not have enough computation resources to meet the application’s performance constraints, which leads to a
high, positive error. By relaxing these constraints, OCCAM provides a 10.2x error reduction in stereo_vision. The stereo_vision benchmark demonstrates how the Contingency Controller tightens the constraints when the system has enough resources to meet those constraints. Between 178s and 398s, the Contingency Controller tightens the constraints because stereo_vision’s compute requirements are lower.

OCCAM’s overhead is fairly low, with the highest overhead being for seismic at 5.79%. This relatively high overhead is the result of the technique used to render only the screen tiles that change. The next largest amount of OCCAM-based overhead is for hearing_aid, at 0.12%. This overhead is mainly due to the quality measurement function, which uses an FFT to analyze the frequency content of the audio data. For all of the other benchmarks, the overhead imposed by OCCAM was too low to be measured as anything higher than 0.00%.

4. RELATED WORK

Application adaptation has been widely studied in the past, and continues to be studied to this day. Many have researched application-specific application adaptation, such as [22], [15], and [14]. While their work shows the usefulness of application adaptation, their methods are essentially ad hoc in the sense that there is not a systematic methodology for adapting the applications.

The Illinois GRACE Project [3] developed a system that allows for adapting applications using cross-layer adaptation. Cross-layer adaptation as described by GRACE involves optimizing the application within both the application itself as well as by simultaneously optimizing the runtime system. They demonstrated this cross-layer adaptation by executing a videoconferencing application which minimized power consumption by trading off encoding complexity for network bandwidth. OCCAM differs from GRACE in two ways. First, OCCAM provides a comprehensive programming framework for developing adaptable applications. Second, OCCAM provides multicores application adaptation.

CMU’s Odyssey [19] project also developed a multitasking system for application adaptation. Odyssey strove to optimize mobile systems by trading off how much work gets done on a remote server versus computed locally on the device. Initially, they used their Coda file system, a distributed file system that provides application-transparent adaptability, and later added application-aware adaptability. Like GRACE, Odyssey focused heavily on adapting applications by trading network bandwidth with local processing. OCCAM differs from Odyssey in that Odyssey provided applications with an interface that informed the application about what resources were available to it. Odyssey did not tackle the actual process of providing a framework for adaptable application development.

Pedersen and Parameswaran [20] present another methodology for application adaptation. In this work, the authors present several different programming techniques for providing application self-adaptation: conditional selection of code and/or functions; and various kinds of loop modification. They then modify several multimedia applications with these techniques to allow the applications to adapt themselves to a fixed energy budget. OCCAM differs from this work in two key ways.

First, OCCAM is not a framework for writing self-adaptable applications; rather, it is a framework for allowing a run-time system to adapt applications. Such an approach is more flexible in that it allows the application to be written once and later adapted to meet different constraints and goals. Second, OCCAM’s adaptation methodology is designed to provide scalable parallel adaptability.

5. SUMMARY AND CONCLUSIONS

This paper presented OCCAM, a development platform to systematically explore and rigorously apply portable, application-specific run-time optimization. By providing facilities to allow the developer to provide information about the performance constraints and application-specific adaptation, OCCAM can provide portability between different computer systems with different resource availabilities.

OCCAM can effectively adapt applications at run time to optimize system resource usage under performance constraints. The studies conducted in this paper show that, compared with a general-purpose, dynamic power management system, OCCAM can improve system energy efficiency by 38% for an Intel Atom-based energy-constrained portable system, and by 24% for a high-performance dual-core computer platform. We also demonstrated that OCCAM can scale to a 16-core system without recompilation. Moreover, we showed that, by leveraging application-specific knowledge, OCCAM offers better online adaptation decisions, resulting in lower energy consumption, faster adaptation, and more successfully meeting application performance constraints. Moreover, OCCAM is very low overhead, with a maximum of 5.79% overhead seen in seismic. Finally, we showed that OCCAM can successfully control applications running on systems with insufficient compute resources by relaxing the application’s performance constraints in a structured manner.

Figure 5: OCCAM vs. powernowd: success at controlling stereo_vision over time on MULTI.
6. REFERENCES


