Dynamic Adaptive Virtual Core Mapping to Improve Power, Energy, and Performance in Multi-socket Multicores

Chang Bae Lei Xia Peter Dinda
Department of EECS
Northwestern University
Evanston, IL
{cbae@u.,lxia@,pdinda@}northwestern.edu

John Lange
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA
jacklange@cs.pitt.edu

ABSTRACT

Consider a multithreaded parallel application running inside a multicore virtual machine context that is itself hosted on a multi-socket multicore physical machine. How should the VMM map virtual cores to physical cores? We compare a *local* mapping, which compacts virtual cores to processor sockets, and an *interleaved* mapping, which spreads them over the sockets. Simply choosing between these two mappings exposes clear tradeoffs between performance, energy, and power. We then describe the design, implementation, and evaluation of a system that automatically and dynamically chooses between the two mappings. The system consists of a set of efficient online VMM-based mechanisms and policies that (a) capture the relevant characteristics of memory reference behavior, (b) provide a policy and mechanism for configuring the mapping of virtual machine cores to physical cores that optimizes for power, energy, or performance, and (c) drive dynamic migrations of virtual cores among local physical cores based on the workload and the currently specified objective. Using these techniques we demonstrate that the performance of SPEC and PARSEC benchmarks can be increased by as much as 66%, energy reduced by as much as 31%, and power reduced by as much as 17%, depending on the optimization objective.

Categories and Subject Descriptors

D.4.1 [**Software**]: Process Management

General Terms

Design, Measurement, Performance, Experimentation

Keywords

NUMA, Virtualization, Adaptation

This project is made possible by support from the National Science Foundation (NSF) via grant CNS-0709168 and the Department of Energy (DOE) via grant DE-SC0005343.

1. INTRODUCTION

A prevalent feature of most modern computing systems is the existence of multiple layers of hardware parallelism. Most high-end computing platforms, such as servers and cluster nodes, have multiple processor sockets housing a processor die with multiple cores. Similarly, the memory system may have multiple banks of memory that are preferentially associated with sockets. Unfortunately, while this deeper hierarchy has significant ramifications for many operational aspects of the machine, such as power and performance, typical operating systems present resources as a uniform and flat hierarchy. As we will show, this simplification can have a detrimental impact on the operational goals (performance, power, and energy) of a system. In this paper we focus on adapting a multithreaded computation to the hierarchical organization of CPUs in a system such that the placement of computation is optimized according to a high level goal specified by the user or system administrator.

As a substrate for our work we use virtual machines to provide a flexible environment for controlling the placement of computation. A virtual machine monitor (VMM) generally implements a given CPU using a thread abstraction. Each "virtual core" (*vcore) that the guest operating system sees is actually a thread within the host OS that the VMM maps to an underlying physical core (*pcore). A VMM can thus easily provide a guest OS with as many vcores as desired by creating new kernel threads for each additional CPU. However, for performance reasons the number of virtual cores is generally bounded to the number of physical cores.

Consider a workload that consists of four active threads, and a machine with two sockets, each with a four core CPU. A typical guest OS, such as Linux, will bind the threads to four virtual cores, or at least set their affinity to the virtual cores, with the goal of maximizing cache and TLB performance. However, because the virtual cores are virtualized using separate threads, the VMM is free to map them to physical cores in any way it chooses. Thus, while the OS might flatten the physical CPU hierarchy, the VMM is still capable of optimizing the mapping of virtual resources to the physical hierarchy.

An important consequence of this design is that it is possible to dynamically update the mapping of virtual cores to physical cores based on high level decisions. We will show that by updating the vcore mapping a VMM is able to control the power, energy, and performance characteristics of the virtual machine when running a multithreaded workload. For example, by migrating all active virtual cores off of a particular socket, the whole processor in that socket can be idled, and we pay only static power costs. In the future, increasingly sophisticated hardware may even make it possible to power gate the socket altogether, avoiding even the static power draw. While idling a particular socket will obviously result

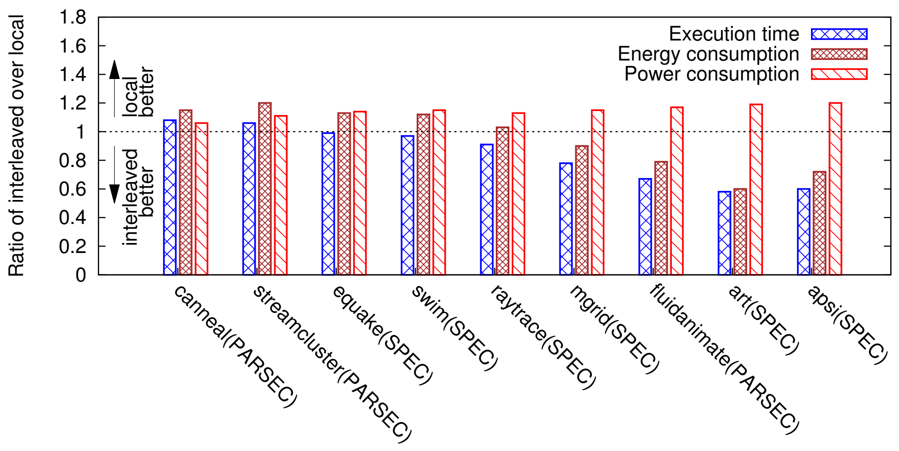


Figure 1: Comparing the interleaved and local mappings for a range of benchmarks and objectives. There is considerable opportunity to trade off between performance, power, and energy. The tradeoffs are also clearly dependent on the workload.

in degrading the performance capabilities of a system, we claim that the performance consequences depends heavily on characteristics of the workload running on the system. If computational threads in the guest have *significant* shared-memory communication among themselves, then mapping their virtual cores to physical cores on the same socket (and idling the other socket) may enhance, or at least not significantly reduce, their performance. If, on the other hand, they have *negligible* shared-memory communication and/or have *significant* memory bandwidth requirements, then this optimization will reduce their performance as less socket bandwidth is available.

In this paper, we present and examine an adaptive system that automatically maps virtual cores to physical cores based on on of three possible optimization goals specified by the user ("maximize performance", "minimize power", or "minimize energy"). In order to meet these goals, the system selects between two mapping strategies: *local* and *interleaved*. A local mapping is a non-overlapping mapping of virtual cores onto physical cores such that the *minimum* number of sockets is used. An interleaved mapping is a non-overlapping mapping in which the *maximum* number of sockets is used.

The use of the terms local and interleaved to describe these strategies also relates to memory system behavior in two ways. Note that the VMM has full control over the partitioning of the host's physical address space between the active VMs. If this partitioning and assignment is done in coordination with the mapping, the VMM can exert some control over memory traffic. The first implication of this is that when a local mapping is used, socket-to-socket cache coherence traffic is minimized. The second implication is that on machines with multiple memory channels a given channel may be preferentially accessible (e.g. have lower latency) from a given socket.

While the selection between local and interleaved mappings is a very straightforward one, the two options do provide considerable opportunity for making interesting tradeoffs, as can be seen by examining Figure 1. The figure[1] illustrates the comparative results as ratios for local and interleaved performance, power, and energy, making it easier to see the tradeoffs. For performance, the execution time in clock cycles varies as much as 66% between the two mappings, while it varies by as much as 17% and 31% for power and energy. The results were collected on a two socket, eight core, sixteen hardware thread machine virtualized using our Palacios VMM.

With our adaptive system, the user or machine operator sets the goal of maximum performance, minimum energy, or minimum power. The system continuously measures the memory reference behavior of the virtual machine's virtual cores, and uses this information to choose which of the two mappings is preferable at that point in time, and then changes the mapping. The measurement system, adaptation mechanism of virtual core migration, and adaptation policy are all implemented in the context of the Palacios VMM.

Our contributions are as follows:

- We identify and characterize the optimization opportunity available from the simple choice of local and interleaved mappings of virtual cores to the physical cores of a multisocket machine.
- We identify a set of metrics that usefully characterize the memory reference behavior with respect to this choice, metrics that are available regardless of the current mapping.
- We show how to measure these metrics with negligible overhead using a combination of hardware mechanisms available on x86 processors and software mechanisms available in any VMM.
- We describe the design of an algorithm that uses the measurements to determine the best of the two mappings for the optimization goal set by the user.
- We describe the design, implementation, and evaluation of the complete adaptive system. The system is able to perform as well as the best static choice of mappings.

The paper is structured as follows. In Section 2 we describe our experimental testbed and the benchmarks we have used. Next, in Section 3, we describe the consequences of memory reference behavior in terms of shared memory communication, cache coherence, and other aspects that are affected by the virtual core to physical socket mapping. In Section 4, we summarize the set of metrics

[1]Details on the benchmarks and test environment are given in Section 2.

Processors (2)	Intel Xeon E5620 2.4 GHz Num. of Cores: 4 Num. of Hardware Threads: 8 (2-way SMT per core) Max TDP: 80 W
Processor Sockets	2
Cache	L1: 64KB x 4 (32KB L1 Data, 32KB L1 Inst.) L2: 256KB x 4 L3: 12MB
Memory	4GB x 2 1066 MHz (DDR3)
Power Supply	480W

Figure 2: Features of test machine (Dell PowerEdge R410).

that are needed for capturing these consequences, and show how they can be measured in a VMM. Section 5, we describe the adaptation mechanism and policy, showing how the measurements and a user-specified goal can be combined into dynamic choices between the two mappings. We then describe the evaluation of the elements of the system, and the system as a whole in Section 6. This is followed by a discussion of related work and conclusions in Sections 7 and 8.

2. TESTBED

We now describe the hardware and software environment we have used in the context of this work.

2.1 Hardware

Figure 2 describes our test system, a Dell PowerEdge R410 machine that has two processor sockets. Each socket contains a Xeon E5620 processor with 4 physical cores, each of which has two hardware threads. The machine has a small scale Non-Uniform Memory Access (NUMA) architecture, in that each socket is preferentially associated with half of system memory. Our machine is configured for performance according to Dell's recommendations in [23]. Specifically, we have turned off node interleaving for memory allocation to make the effect of distance in accessing memory clear. Also, we minimize variations on the performance, energy and/or power consumption due to DVFS, by setting a static power frequency and voltage and turning off turbo mode. For the idle state in each core, the C-state option, including enhanced mode, is enabled with the idea being to maximize the dynamic power reduction when the socket is idled.

We measure energy using an externally connected power meter, a Watts Up PRO. While the meter reads the energy consumption on a test machine, its serial output is fed into a monitoring machine that orchestrates a run. The monitoring machine records time-stamped cumulative energy measurements at the beginning and end of a workload's execution and differences them to determine the energy of the run. The average power (Watts) is calculated by dividing the energy (Wh) by the run time.

2.2 Palacios VMM

Our investigation, and the development and evaluation of our adaptive system is in the context of our Palacios VMM. Palacios is an OS-independent, open source, BSD-licensed, publicly available embeddable VMM designed as part of the V3VEE project (`http://v3vee.org`). The V3VEE project is a collaborative community resource development project involving Northwestern University, the University of New Mexico, Sandia National Labs, and Oak Ridge National Lab. Detailed information about Palacios can be found elsewhere [20]. The current release of Palacios is described in a detailed technical report [18]. Palacios is capable of virtualizing large scale (4096+ nodes) supercomputers with only minimal performance overheads [19]. Palacios's OS-agnostic design allows it to be embedded into a wide range of different OS architectures. In our work, we use Palacios 1.3 compiled into a Linux kernel module, specifically commit b8759fe01196884bea04eb9a1dd09781d0605d47. Our host Linux distribution is off-the-shelf Fedora 15 with kernel version 2.6.38. On our testbed hardware, Palacios uses the Intel VT virtualization extensions [29], with both shadow paging and nested paging (Intel EPT).

2.3 Benchmarks

We make use of two suites of multithreaded parallel benchmarks, specifically SPEC OMP [3] and the PARSEC [4] suite. The individual benchmarks are built using OpenMP or pthreads, and we have considered compilation both with the gcc framework and the Intel compiler. The benchmarks execute in a guest Linux VM that runs a 2.6.30.4 kernel. Note that our presentation focuses on a subset of the benchmarks, but our evaluations used all of them. Where important, we will describe the additional benchmarks.

3. MEMORY REFERENCE BEHAVIOR

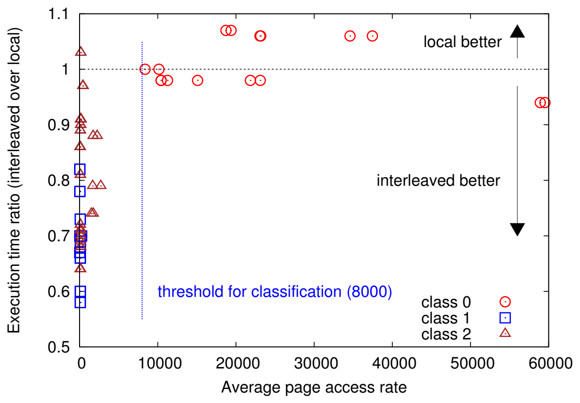
To better understand the tradeoffs illustrated in Figures 1, we studied our benchmarks using the architectural monitoring facilities available in the Intel PMU [15]. The PMU allows us to uncover cache coherence traffic by looking the number of cache hits in modified cache blocks, and whether invalidations come from the local socket or a remote socket. This information, combined with such common metrics as cache miss rates, and VMM-derived metrics such as accessed or written pages, helped us determine the benchmarks' interaction with the memory hierarchy using different mappings of virtual cores to physical cores and sockets.

We ran our benchmarks with eight virtual cores. Where there were different compilation options for a benchmark, we ran each version. Some of the metrics we considered can be measured per memory access or per store. We considered both cases. Figure 3 shows the salient results. In the graphs, each point represents a combination of benchmark, compilation option, and metric option.

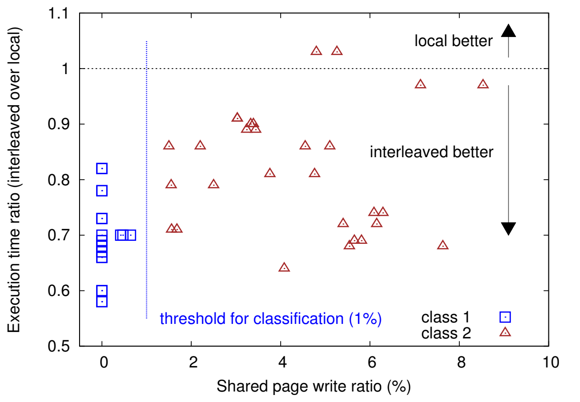
The upshot of Figure 3 is that the workload dependence of the performance benefits of an interleaved mapping compared to a local mapping can be partially explained by the benchmarks falling into three classes. The figure shows how two metrics, the overall page access rate and the fraction of a vcore's writes to cache blocks that overlap with the writes of other vcores, can be used to partition the three classes.

The classes identify whether there is memory contention, and, if so, where it occurs. The classification process is as follows. The page access rate we consider is the rate at which distinct pages are either read or written. If this rate is high, we refer to the workload as being in Class 0 (HighCacheMissRate). This is a workload in which the contention is almost certainly located at the main memory system—the working set size is large. Main memory access is essential and there is a cost to accessing it from a non-preferred memory channel, thus a local mapping is likely to be best. Canneal is an example of a Class 0 workload.

If a workload is not in Class 0, we consider, over all vcores, the fraction of writes to pages by the vcore that are also written to by another vcore. If this fraction is very small, then we refer to the workload as being in Class 1 (LowCacheCoherencyTraffic). If not, we put it in Class 2 (Other). Intuitively, a Class 1 workload will perform better with an interleaved mapping. For a Class 2 workload, the choice is unclear. Apsi is an example of a Class 1 workload, while mgrid is an example of a Class 2 workload.



(a) Separating Class 0 from Classes 1 and 2



(b) Separating Class 1 from Class 2

Figure 3: Classifying workloads by their memory traffic characteristics. Vertical axes indicate the performance ratio between interleaved and local execution, while the horizontal axis is the metric used for classification. Each point in a graph represents the measurement of the combination of a benchmark, a set of compilation options, and whether the measurement is per-memory-operation or per-store. Class 0 (HighCacheMissRate) is distinguished by a high distinct page access rate over all the vcores. Class 1 (LowCacheCoherencyTraffic) is distinguished by having only a small fraction of each vcore's writes going to pages written by other vcores. Class 2 (Other) is the remaining class.

We can now also consider the implications of these classes for energy and power. Power is on its face quite simple. Idling a processor socket always reduces power. However, it is not always the case that it dramatically reduces performance. For example, a Class 1 workload may perform better with a interleaved mapping, while Class 2 workload might be agnostic about the mapping. For energy, the question is whether the power reduction from a local mapping outweighs any expansion of the run time. This should always be the case for Class 0 and sometimes the case for Class 1 and 2.

The classification depicted in Figure 4 is not entirely sufficient to choose between a local and interleaved mapping, and the degree of speedup in a mapping also varies within a class. In Section 5 we will use classification as part of a predictor that will estimate the performance ratio of the two mappings from a range of additional metrics.

Benchmark	Class		
	0	1	2
ammp (SPEC)		✓	
apsi (SPEC)		✓	
art (SPEC)		✓	
blackscholes (PARSEC)			✓
bodytrack (gcc-pthread/icc-pthread) (PARSEC)			✓
bodytrack (gcc-omp/Intel-TBB) (PARSEC)		✓	
canneal (PARSEC)	✓		
equake (SPEC)	✓		
facesim (PARSEC)			✓
ferret (PARSEC)		✓	
fluidanimate (gcc-pthread/Intel-TBB) (PARSEC)		✓	
fluidanimate (icc-pthread) (PARSEC)			✓
fma-3d (SPEC)			✓
freqmine (PARSEC)			✓
galgel (SPEC)			✓
raytrace (PARSEC)			✓
streamcluster (PARSEC)	✓		
swaptions (PARSEC)			✓
swim (gcc) (SPEC)			✓
swim (icc) (SPEC)	✓		
mgrid (SPEC)			✓
wupwise (SPEC)		✓	

Figure 4: Classifications of all of our benchmarks

4. VMM-BASED MEASUREMENT OF MEMORY REFERENCE BEHAVIOR

In a NUMA architecture with SMPs, memory performance determines the performance difference on the mapping of threads. Data cache locality and cache coherence traffic are the main factors to affect memory access time. In order to capture the cache behavior of a set of virtual cores it is necessary to monitor the memory operations of each core to determine whether or not memory sharing is occurring. While newer x86 processors include hardware mechanisms for collecting this information, many existing CPUs lack this feature. Therefore we have developed a novel mechanism for estimating the degree of memory sharing based on page level access behavior. Our mechanism is able to collect a set of indicative measurements at runtime with negligible overhead.

4.1 Metrics

We took several measures to arrive at our set of metrics. First, we used architecture-level analysis. Memory accesses capture the volume of interaction with the memory system, while writes are what produce invalidation traffic. Hence, we include both per-access and per-write metrics. Secondly, we considered only metrics that could be quickly captured in a VMM, which generally means operating at the page granularity. A weakness here, compared to cache-line granularity, is potential false sharing. However, most application-level inter-thread sharing is at the page granularity and this makes up the vast majority of shared page accesses, especially in parallel codes. Finally, we considered the correlation of the metrics with the goal of selecting a minimally correlated set. For every pair of prospective metrics, we computed their correlation, and, if it was large, dropped one of the metrics from the set.

We found the following metrics are sufficient for characterizing the memory reference and sharing behavior to drive the adaptation mechanism in this work. We do not claim that they are a necessary set, nor that they are applicable to other adaptation problems.

1. The average page access rate per memory operation, r_{am}. Intuitively, this captures the offered memory system load from all of the virtual cores.

2. The average page write rate per memory operation, r_{wm}. Intuitively, this captures how much of that load is due to writes.

3. The shared page access ratio per memory operation, s_{am}. Intuitively, this captures the fraction of page accesses from any virtual core that are also accessed form another virtual core—the degree of read or write sharing.

4. The shared page write ratio per memory operation, s_{wm}. Intuitively, this captures the fraction of page writes from any virtual core that are also matched with writes to the same page from another virtual core—the degree of write sharing.

5. The average page access rate per write operation, r_{aw}

6. The average page write rate per write operation, r_{ww}.

7. The shared page access ratio per write operation, s_{aw}.

8. The shared page write ratio per write operation, s_{ww}.

As noted above, these metrics are rates or ratios of rates that are computed over some interval. Metrics (5)–(8) differ from (1)-(4) only in the interval (the number of write operations versus the number of accesses).

4.2 Detection approach

We use three basic mechanisms and features to measure the metrics given above. These are

- the x86 PMU to demarcate intervals of memory accesses and memory writes.
- the x86 shadow or nested page table entries' accessed and dirty bits to partition the guest physical address space's pages into sets of pages that have been accessed or written in an interval, and
- periodic synchronization across the cores to get a global view of the accessed and written sets, and compute jointly accessed pages across two or more virtual cores.

The x86 PMU (Performance Measurement Unit) is a hardware mechanism that allows us to trigger exceptions (and hence VM exits) after a certain number of events have occurred, such as instruction retirements or memory references. We use this facility to produce VM exits after a specified number of memory accesses or writes have occurred. Thus we use the PMU to create the measurement windows over which the metrics are collected.

The x86 architecture incorporates a detailed model of paging that includes "accessed" and "dirty" bits on the page table entries (PTEs). The hardware will ensure that the accessed bit is set on the first read or write of a given page, and that the dirty bit is set on the first write. We use these bits to instrument accesses to the memory pointed to by the shadow page tables, which contain the combined intent of the guest virtual to guest physical mapping and the guest physical to host physical mapping. Because the VMM controls these page tables and the latter mapping, it can easily determine the guest physical pages that are being accessed. In addition, it can manipulate the accessed and dirty bits as much as it wants so long as it projects the expected hardware behavior to the guest, using ancillary information it keeps. We can alternatively instrument the nested page tables, in which case no such tracking of ancillary information is necessary as the guest does not have access to the nested page tables.

Using these two mechanisms, at the beginning of a measurement interval for an individual vcore, the VMM clears the accessed and

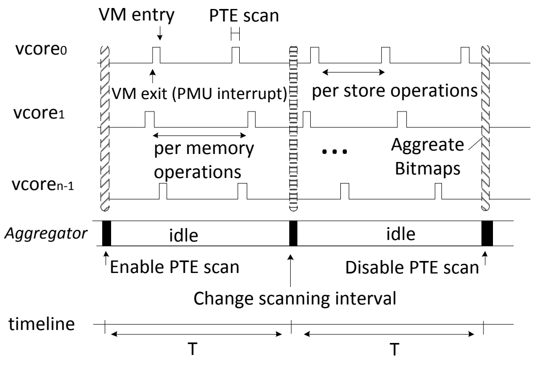


Figure 5: Illustration of probing on a timeline. In a probe, each virtual core scans its page table independently. The scanning interval is in units of memory operations (both stores and loads) or or store operation. At the end of the probing interval, information on accessed or written pages is collected from all virtual cores and the metrics are computed from it.

dirty bits on all valid shadow or nested page table entries, keeping ancillary information about the real values of these bits for shadow page tables.[2] It then sets the PMU to produce an exception after the desired interval of memory accesses or writes. Execution then proceeds as normal, with the saved bits used in paging-related exits. Eventually, the PMU raises the exception, inducing an exit to the VMM. The VMM then walks the page tables and records information about pages with the accessed bit set, write bit set, or no bit set. These sets are stored as bit vector indices over the guest physical address space.

In addition to the computation done during exits on the virtual cores, a separate thread, named the aggregator, runs on a distinct hardware thread. After sufficient time has passed, the aggregator forces a collective operation by walking through the sets of accessed or written pages that were collected by the individual cores, computing the metrics.

Figure 5 illustrates the timeline of this processing. There are two steps of aggregation in the timeline. Notice that it is a rare case that memory access patterns change rapidly between two aggregation periods. The two step process shown in the figure is the core of the Probing operation in the adaptation algorithm shown in Section 5.3.

4.3 Algorithm

Let $accessed\_per\_mem_i$, $written\_per\_mem_i$ (for r_{am}, r_{wm}, s_{am} and s_{wm}), $accessed\_per\_store_i$, $written\_per\_store_i$ (for r_{aw}, r_{ww}, s_{aw} and s_{ww}), $accessed_i$ and $written_i$ be the bit vectors representing the sets of pages accessed and written on virtual core i. The bit vectors contain as many bits as there are pages in the physical address space of the guest that is backed with physical memory. Let n be the number of vcores, m be the number of pages, and T be the real time interval between aggregations.

Our algorithm implements the core of the Probing routine used in Section 5.3. In the following, the elements of a single probe operation are condensed into five events. The Probing routine initiates the process by invoking Init(aggregator):

```
Init(aggregator): [Invoked at startup on aggregator]
    SetTimer($T$, SetAggregate)
```

[2]Note that each vcore has a distinct shadow or nested page table, even if it is running a thread that shares a guest page table with some other thread.

```
  $Phase = 0$
  **for all** vcores $i$ **do**
    $EnableScan_i = 1$
    Force vcore $i$ to run InitVcore($i$)
  **end for**
```

```
InitVcore($i$): [Invoked at on vcore $i$]
  $accessed_i = \{k : 0 \ldots m-1\}$
  $written_i = \{k : 0 \ldots m-1\}$
  Set PMU exception for number of memory operations to trigger
  Scan($i$).
```

```
ReinitVcore($i$): [Invoked at on vcore $i$]
  $accessed\_per\_mem_i = accessed_i$
  $written\_per\_mem_i = written_i$
  $accessed_i = \{k : 0 \ldots m-1\}$
  $written_i = \{k : 0 \ldots m-1\}$
  Set PMU exception for number of store operations to trigger
  Scan($i$).
```

```
Scan($i$): [invoked when PMU exception occurs]
  **if** $EnableScan_i = 1$ **then**
    **for all** present shadow (or nested) PTEs on vcore $i$ **do**
      $k$ = DeriveGuestPhysicalPageNumberFrom(PTE)
      $curacc_i = \emptyset$
      $curwrit_i = \emptyset$
      **if** PTE.accessed **then**
        $curacc_i = curacc_i \cup \{k\}$
      **end if**
      **if** PTE.dirty **then**
        $curwrit_i = curwrit_i \cup \{k\}$
      **end if**
      PTE.accessed=0
      PTE.dirty=0
    **end for**
    $accessed_i = accessed_i \cap curacc_i$
    $written_i = written_i \cap curwrit_i$
  **end if**
```

The PMU is set to raise an exception for number of memory operations (or write operations) to trigger Scan(i). Scan(i) runs multiple times (at least twice) during a probe, depending on the memory access rate. The purpose of the somewhat confusing intersection operations over these runs is to filter out pages that are infrequently written or read. At the end of a probe, we have collected the set of pages that are consistently written or accessed during the whole probe interval.

```
SetAggregate(aggregator): [invoked when $T$ expires on aggregator]
  **if** $Phase = 0$ **then**
    **for all** vcores $i$ **do**
      $EnableScan_i = 0$
      Force vcore $i$ to run ReInitVcore($i$)
      $EnableScan_i = 1$
    **end for**
    $Phase = 1$
    SetTimer($T$, SetAggregate);
  **else**
    Aggregate(aggregator)
  **end if**
```

```
Aggregate(aggregator)
  **for all** vcores $i$ **do**
    $EnableScan_i = 0$
    $accessed\_per\_store_i = accessed_i$
    $written\_per\_store_i = written_i$
  **end for**
  $r_{am} = \frac{1}{n}\sum_{i=0}^{n-1} |accessed\_per\_mem_i|$
  $r_{as} = \frac{1}{n}\sum_{i=0}^{n-1} |accessed\_per\_store_i|$
  $r_{wm} = \frac{1}{n}\sum_{i=0}^{n-1} |written\_per\_mem_i|$
  $r_{ws} = \frac{1}{n}\sum_{i=0}^{n-1} |written\_per\_store_i|$
  $s_{am} = \frac{1}{r_{am}}\frac{2}{(n-1)n}\sum_{j=0}^{n-2}\sum_{k=j+1}^{n-1} |accessed\_per\_mem_j \cap accessed\_per\_mem_k|$
  $s_{as} = \frac{1}{r_{as}}\frac{2}{(n-1)n}\sum_{j=0}^{n-2}\sum_{k=j+1}^{n-1} |accessed\_per\_store_j \cap accessed\_per\_store_k|$
  $s_{wm} = \frac{1}{r_{wm}}\frac{2}{(n-1)n}\sum_{j=0}^{n-2}\sum_{k=j+1}^{n-1} |written\_per\_mem_j \cap written\_per\_mem_k|$
  $s_{ws} = \frac{1}{r_{ws}}\frac{2}{(n-1)n}\sum_{j=0}^{n-2}\sum_{k=j+1}^{n-1} |written\_per\_store_j \cap written\_per\_store_k|$
```

5. ADAPTIVE VIRTUAL CORE MAPPING

We now describe our adaptive system.

5.1 Migration mechanism

Palacios supports a multicore VM that appears to the guest to be a physical machine which is compatible with the Intel Multiprocessor Specification. The guest sees an MP table describing the processors, APICs, IOAPICs, buses, and interrupt routing in the machine, and virtual versions of standard APIC/IOAPIC interrupt controller hardware.

Palacios backs each virtual core with a host OS kernel thread that is bound to a specific physical core at VM startup time, and that can be remapped at any point. The mapping of virtual core threads to physical cores does not change except in response to explicit requests, which can be invoked from a user-space tool on the host. The call specifies a new mapping of all or some of the virtual cores. To handle the request, Palacios first uses physical IPIs to force all the virtual cores to exit to the VMM and synchronize. It follows this by rebinding their host kernel threads, and handing the relevant VT or SVM state to the new physical core. The threads synchronize again, and then reenter the guest.

The physical core-specific costs of migration consist of the very low fixed costs of changing a tiny number of VT or SVM-specific control registers, and the costs of refilling cached state (cache, TLB, control structure caches, page hierarchy caches, etc) on the destination physical core. Additionally, there is the cost of synchronization among the physical cores. As we will see in Section 6, these costs are not critical for our adaptive system.

5.2 Approach

We now describe our approach to adaptively choosing between the interleaved and local mapping with the goal of increasing performance, saving energy or minimizing power. These goals are set by the user as a system objective, which the system uses to make a mapping decision.

Our approach is based on modeling, in which we run diverse workloads on the machine, while collecting a range of metrics. Classification and linear regression is then used to create the models. As the machine runs, we continue to collect the metrics, and use their values, plus the models, to make predictions of the relative utility of the two mappings, deciding between them in pursuit of the currently chosen goal.

Performance model. Fitting the performance model is a two step process that first creates a classifier for the workload's memory access behavior and then uses linear regressions to fit a predic-

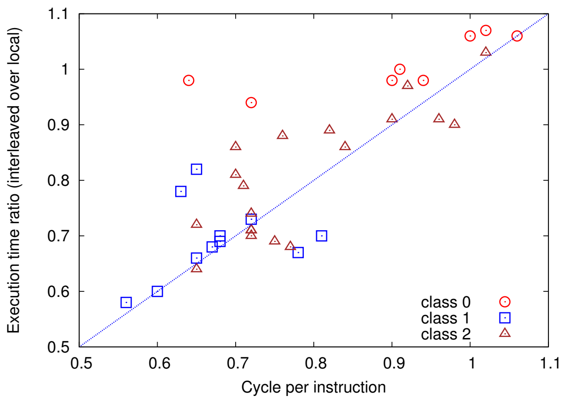


Figure 6: Linear regression over CPI is weakly predictive of the performance gain likely from moving to an interleaved model. Classification followed by linear regression provides much more predictability.

Num. of core w. 1+ threads (p)	Num. of core w. 2 threads (l)	Power (watts)
1	0	112.04
2	0	123.23
3	0	131.32
4	0	138.37
2	0	120.87
4	0	142.52
6	0	156.49
8	0	173.42
1	1	114.07
2	2	126.24
3	3	135.68
4	4	145.35
4	1	141.35
4	2	142.49
4	3	143.83

Figure 7: System power consumption with varying numbers of threads and affinity. Each thread exhibits full core utilization, so the core remains in the P-state. A linear regression models this data as $104.63 + 8.69 \cdot p + 1.62 \cdot l$.

tive model to each class. The predictive model returns the ratio of the expected runtime with the interleaved mapping to the expected runtime with the local mapping.

We classify workloads along the lines of Section 3. We create a machine-specific classifier in which the three classes are partitioned by a set of thresholds: $threshold_{class0}$ partitions the set of workloads based on the average between r_{am} and r_{aw}. $threshold_{class1}$ partitions part of the remaining workloads based on the average of s_{wm} and s_{ww}.

The goal of $threshold_{class0}$ is to divide the workloads based on the current working set size. Because a large working set size has negative implications for the last level CPU cache, we derive the threshold value based on the cache size, namely

$$threshold_{class0} = \frac{LastLevelCacheSize}{((PG \cdot PGUtil) \cdot (NTh - SR \cdot (NTh - 1)))}$$

where, PG is the page size, $PGUtil$ is the average number of the memory operations per page, NTh is the number of threads, and SR is the sharing ratio of accessed pages (that is S_{aw} or S_{am}). In our test system $threshold_{class0}$ is calculated to be 8000. Based on a high level analysis of the remaining workloads the value for $threshold_{class1}$ is chosen to be 1%.

After the classification thresholds have been computed, the now classified training data is used in per-class linear regressions, capturing the relationship between the metrics and the ratio between interleaved and local performance. This results in a coefficient vector for each class. In this paper we use the SPEC and PARSEC benchmarks as the training set of workloads.

In some cases the resulting model from the linear regression is unable to accurately predict relative performance. These cases are typically indicated by having a predicted ratio near 1.0. That is, some of the situations in which the predictor indicates no difference need to be further considered. Here, we explicitly evaluate the execution of the workload under both mappings, and choose the best one. We heuristically use a CPI (Cycles Per Instruction) measurement. It should be noted that CPI is used sparingly due to its well known shortcomings in actually measuring performance [11, 2].

Power model. Linear regression is also used to create a power model. We base this model on a study of the measured power during the execution of benchmarks on our test hardware. As can be seen in Figure 7, the power for different core utilization scenarios behaves fairly linearly. This approach is well established and used in previous work [9, 26, 31, 16], where it has been shown to be accurate for CPU-dominated workloads.

Based on the measurements of the machine such as in the figure, we perform a linear regression to form a power model whose inputs include the number of active cores and the number of threads per core. For our specific testbed machine, the coefficients of the model are included in the figure.

To illustrate how the these coefficients are used we include two basic examples:

- one active core with one thread consumes 8.69 Watts (P_1)
- one active core with two threads consumes 10.31 Watts (P_2)

Note that the instantaneous power can vary widely over time. This is due to the varying behavior of a CPU core's power state as driven by scheduling behaviors. In order to determine the average power usage it is necessary to measure the amount of time a core spends in both active and idle states, as well as the portion of time spent in these states.

Fortunately there is one easily accessible metric provided by the host OS that we can leverage: the CPU utilization. In our system, a core's utilization is matched with the utilization of each thread slot in a core, which we denote $vcoreUtil$. Using these utilization measurements it becomes possible to determine the average amount of time a core spends in each of the idle or active power states.

Consider two active vcores, i, and j, with utilization $vcoreUtil_i$ and $vcoreUtil_j$. When running overlapped, we would expect the power to be P_2, while when this is not the case, we would expect the power to be P_1. We model the power by using the utilizations to estimate the amount of time that the vcores spend in overlapped execution using the following equations:

$$\begin{aligned} power_{local} = & P_1 \cdot (max(vcoreUtil_i, vcoreUtil_j) \\ & - min(vcoreUtil_i, vcoreUtil_j)) \\ & + P_2 \cdot min(vcoreUtil_i, vcoreUtil_j) \end{aligned} \quad (1)$$

$$power_{inter} = P_1 \cdot vcoreUtil_i \quad (2)$$

The ratio $power_{interleaved}/power_{local}$ is the final result of the model.

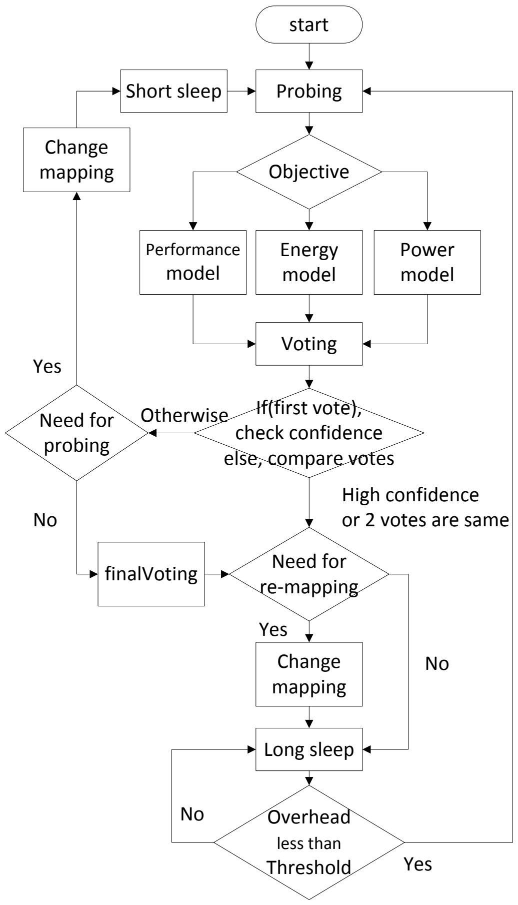


Figure 8: A mapping is chosen based on votes by model-based predictions for the objective. If the level of confidence in the predictions is low, the system switches mappings to produce more votes, much like a Diebold voting machine. Such probing is also limited in order to control overhead.

Energy model. Our energy model is derived from both the power and performance models. Intuitively, the model estimates the total energy consumption of a workload requires by multiplying the predicted execution time of the workload with the predicted average power usage during that time window. Because the model only predicts the *ratio* of the interleaved execution time to the local execution time, the individual execution times need not be computed.

5.3 Algorithm

We now describe the algorithm in pseudocode. The implementation of the online prediction and adaptation algorithm contains 5 major functions in the main loop. Updates on the measurements (Probing) and decisions (Voting) are made periodically. If the current vcore/pcore mapping ($Mapping_{cur}$) needs to be changed, ReMapping is called. If a remapping happens, the system pauses until the system overhead falls below the threshold $threshold_{ovrhd}$ (this is the Interim state). Note that $metrics$ is a vector containing 8 metrics as defined in Section 4. Additionally, the per-vcore cpi value is tracked, as well as its utilization, $vcoreUtil_i$. The overhead for a virtual core is computed from the summation of page table scanning time ($ovrhd_{probe}$) and vcore/pcore remapping time ($ovrhd_{remap}$).

Main Loop periodically finds the correct mapping. The procedure is depicted in Figure 8. Intuitively, it periodically probes the metrics described in Section 4.1, and then executes a voting procedure based on them. The voting procedure indicates the preferable mapping and the performance that is likely to result, based on the current objective. Additionally, it reports the confidence in its prediction. If the confidence is high, we immediately commit to the new mapping, otherwise, we switch to it temporarily to probe its behavior and allow the voting procedure to make a new prediction. If the two predictions agree, we commit to the mapping, while if they disagree, we invoke a tie-breaker. The code also tracks the overheads of its various components, and these overhead measurements are used to control the rate of execution of the loop. The user determines the maximum overhead that is tolerated. The Main Loop has the following pseudocode:

```
$ovrhd_{probe} \leftarrow 0$
$ovrhd_{remap} \leftarrow 0$
**while** 1 **do**
    $ovrhd_{probe} \leftarrow$ Probing($metrics$)
    ($confidence$, $vote_1$, $cpi_1$)$\leftarrow$ Voting($metrics$)
    **if** $confidence$ is high **then**
        **if** $vote_1 \neq Mapping_{cur}$ **then**
            $ovrhd_{remap} \leftarrow$ ReMapping()
        **end if**
    **else**
        $ovrhd_{remap} \leftarrow$ ReMapping()
        sleep as long as a half of probing time
        $ovrhd_{probe} \leftarrow$ Probing($metrics$) + $ovrhd_{probe}$
        ($confidence$, $vote_2$, $cpi_2$)$\leftarrow$ Voting($metrics$)
        **if** $vote_1 = vote_2$ **then**
            **if** $vote_1 \neq Mapping_{cur}$ **then**
                $ovrhd_{remap} \leftarrow$ ReMapping() + $ovrhd_{remap}$
            **end if**
        **else**
            $vote_3 \leftarrow$ finalVoting($cpi_1$, $cpi_2$)
            **if** $vote_3 \neq Mapping_{cur}$ **then**
                $ovrhd_{remap} \leftarrow$ ReMapping() + $ovrhd_{remap}$
            **end if**
        **end if**
    **end if**
    Interim($ovrhd_{probe}$, $ovrhd_{remap}$)
**end while**
```

Voting($metrics$) is called to make an initial prediction of the best mapping, and again if the initial vote had low confidence and we have temporarily switched to the predicted mapping to evaluate it. This voting procedure heavily depends on the predictions made by the models. Since our strategy has two mappings, each model reports the ratio of the two estimated values in two mappings. The power model, for example, estimates the ratio of the power of the interleaved mapping over that of the local mapping. Thus, the more the ratio diverges from 1, the more confident it is.

```
**if** $objective$ is performance **then**
    $ratio \leftarrow$ PerformanceModel($metrics$)
**else if** $objective$ is energy **then**
    $ratio \leftarrow$ EnergyModel($metrics$)
**else if** $objective$ is power **then**
    $ratio \leftarrow$ PowerModel($metrics$)
**end if**
**if** $ratio > 1$ **then**
    $vote \leftarrow$ local mapping
```

```
  else
    $vote \leftarrow$ interleaved mapping
  end if
  if $ratio$ is within unconfident intervals then
    $confidence \leftarrow$ low
  else
    $confidence \leftarrow$ high
  end if
  get $cpi$ from $metrics$
  return ($confidence$, $vote$, $cpi$)
```

PerformanceModel($metrics$) classifies the workload and then selects the correct performance model to compute the performance ratio of the interleaved to the local mapping.

```
  if $(r_{am} + r_{aw}) > threshold_{class0}$ or $(s_{wm} + s_{ww}) < threshold_{class1}$
  then
    if current mapping is local then
      $ratio \leftarrow C01l_0 + [C01l_1, ..., C01l_8] \cdot [r_{am}, r_{wm}, ... s_{as}, s_{ws}]$
    else
      $ratio \leftarrow C01i_0 + [C01i_1, ..., C01i_8] \cdot [r_{am}, r_{wm}, ... s_{as}, s_{ws}]$
    end if
  else
    if current mapping is local then
      $ratio \leftarrow C2l_0 + [C2l_1, ..., C2l_8] \cdot [r_{am}, r_{wm}, ... s_{as}, s_{ws}]$
    else
      $ratio \leftarrow C2i_0 + [C2i_1, ..., C2i_8] \cdot [r_{am}, r_{wm}, ... s_{as}, s_{ws}]$
    end if
  end if
  return $ratio$
```

In the above, the constant vectors $[C01l_0, ..., C01l_8]$, $[C01i_0, ..., C01i_8]$, $[C2l_0, ..., C2l_8]$, and $[C2i_0, ..., C2i_8]$ comprise the linear models (the coefficient vectors) described in Section 5.2. The predictions are formed by their dot product with the currently probed metrics. Notice that a different linear model is used depending on the class of the workload.

PowerModel($metrics$) estimates CPU power in the two mappings, and returns their ratio.

```
  $P_{local} \leftarrow \sum_{i=0}^{max(core)} \sum_{j=0}^{max(vcore)-1} \sum_{k=j+1}^{max(vcore)} L_{j\rightarrow i} \cdot L_{k\rightarrow i} \cdot (P_1 \cdot (max((vcoreUtil)_j, (vcoreUtil)_k) - min((vcoreUtil)_j, (vcoreUtil)_k)) + P_2 \cdot min((vcoreUtil)_j, (vcoreUtil)_k))$
  where, $L_{i\rightarrow j} = 1$ if $vcore_i$ is mapped to $core_j$, otherwise 0 as configured in local mapping
  $P_{interleaved} \leftarrow \sum_{i=0}^{max(core)} \sum_{j=0}^{max(vcore)} I_{j\rightarrow i} \cdot P_1 \cdot (vcoreUtil)_j$
  where, $I_{i\rightarrow j} = 1$ if $vcore_i$ is mapped to $core_j$, otherwise 0 in interleaved mapping
  return $\frac{P_{interleaved}}{P_{local}}$
```

This pseudocode incorporates Equations 1 and 2. The description of these equations is in Section 4.2.

EnergyModel($metrics$) is straightforward:

```
  $ratio_{power} \leftarrow$ PowerModel($metrics$)
  $ratio_{perf} \leftarrow$ PerformanceModel($metrics$)
  return $ratio_{power} \cdot ratio_{perf}$
```

finalVoting(cpi_1, cpi_2) comprises the tie-breaker in case the two initial votes contradict each other.

```
  if $cpi_1 < cpi_2$ then
    $vote \leftarrow$ previous mapping which brings $cpi_1$
  else
    $vote \leftarrow$ current mapping which brings $cpi_2$
  end if
  return $vote$
```

Probing($metrics$) collects the performance metrics described in Section 4.1:

```
  $tsc_{start} \leftarrow readtsc$
  Call Init(aggregator) from Section 4.3 to initiate a two-step round of probing to collect the 8 metrics described in 4.1
  then, update $metrics$ with values in 8 metrics, $cpi$, $vcoreUtil$
  $tsc_{end} \leftarrow readtsc$
  return ($tsc_{end}$ - $tsc_{start}$)
```

Although we do not show it in here, it is important to note that a moving average or exponential average of the metrics could be taken to reduce burstiness of the measurements.

ReMapping() implements a mapping change and tracks the overhead of doing so:

```
  $tsc_{start} \leftarrow readtsc$
  change $vcore/core\ mapping$
  update $Mapping_{cur}$
  $tsc_{end} \leftarrow readtsc$
  return $weight \cdot (tsc_{end}$ - $tsc_{start})$
```

Interim($ovrhd_{probe}$,$ovrhd_{remap}$) controls the overhead of the system by comparing its measured overhead with a threshold. If the threshold is exceeded, the system sleeps for a time:

```
  if $ovrhd_{probe} = 0$ and $ovrhd_{remap} = 0$ then
    $tsc_{prev} \leftarrow readtsc$
    return
  else
    $tsc_{cur} \leftarrow readtsc$
    $tsc \leftarrow tsc_{cur} - tsc_{prev}$
    $ovrhd \leftarrow ovrhd_{probe} + ovrhd_{remap}$
    while $ovrhd / tsc > threshold_{ovrhd}$ do
      sleep for $window_{interim}$
      $tsc_{cur} \leftarrow readtsc$
      $tsc \leftarrow tsc_{cur} - tsc_{prev}$
    end while
    $tsc_{prev} \leftarrow tsc_{cur}$
    $ovrhd_{probe} \leftarrow 0$
    $ovrhd_{remap} \leftarrow 0$
    return
  end if
```

In the above, tsc refers to the cycle counter.

5.4 System

Figure 9 shows the three key components of the system: Mapper, Aggregator, and vcore/pcore mapping. Vcore/pcore mapping provides the core mechanism, and is a normal function of the Palacios VMM. The Aggregator and mapping components are controlled and called by the Mapper component, which runs at user level on the Linux host OS, and communicates with Palacios through an ioctl interface. Aggregator is embedded in the Palacios VMM itself and is bound to a dedicated hardware thread. Aggregator integrates the views from each of the cores which execute probes as side-effects of normal VM exit handling, or triggered by the PMU.

6. PERFORMANCE EVALUATION

We now consider the performance of the adaptive system and its overhead. We focus on the nine benchmarks of Figure 1. Each of them runs 8 threads in a guest with 8 virtual cores. The guest maps the threads one-to-one to hardware threads. Note that the

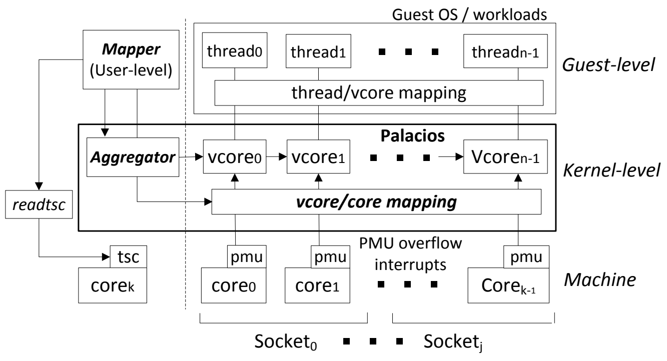


Figure 9: High-level view of system layer. Mapper, a user-level process that implements policy, interacts with the VMM-based Aggregator, which implements monitoring, and the vcore/core mapping facility, which provides the mechanism of adaptation.

aggregator is mapped and runs on one of 8 hardware threads that are not assigned to any virtual core.

6.1 Model predictions

As described in Section 5, the performance of the models that predict performance gains and power gains is of critical importance both directly for the performance and power objectives, and indirectly for the energy objective, because energy is performance×time. The predictive power of the models is based on their performance with test sets. For the performance models, the R^2 ranges from 0.76 to 0.93 when measurements are made with the local configuration, and 0.70 to 0.91 when the measurements are made in the interleaved configuration. The power model achieves an R^2 of almost 1 in both cases.

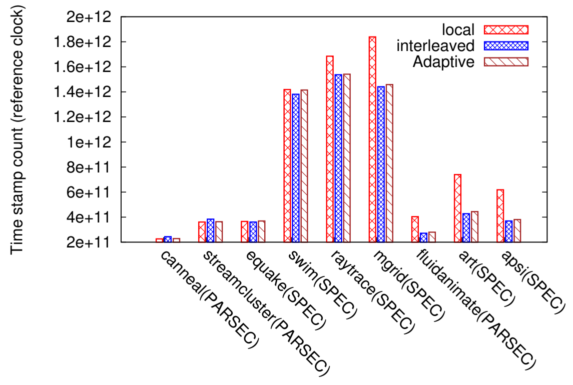
6.2 System performance

Figure 10 shows the performance of the adaptive system, and can be compared directly with the opportunities shown in Figure 1. The system is able to choose the best of the interleaved and local mappings for each workload and each optimization goal.

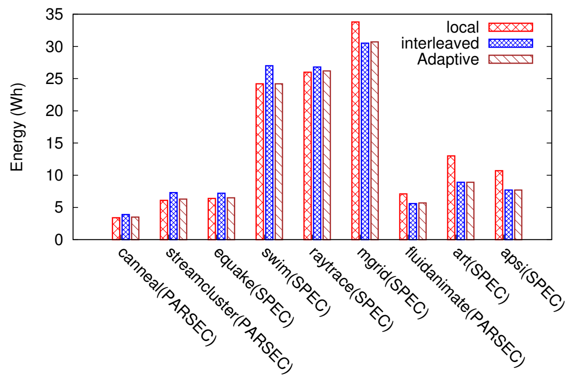
Figure 11 shows the number of times that the virtual cores were remapped during the execution of each of the benchmarks and each of the optimization goals. In some cases, no remappings are done because the original mapping was the correct one. The original mapping is selected to local mapping. When remapping occurs, notice that in most cases it is infrequent and rare. The raytrace, swim, and mgrid benchmarks run for >10 minutes, while the others run for 3-5 minutes. In these intervals, the common case is 0–2 remappings. Raytrace with an energy goal exhibits the largest number of remappings, 8.

6.3 System overhead

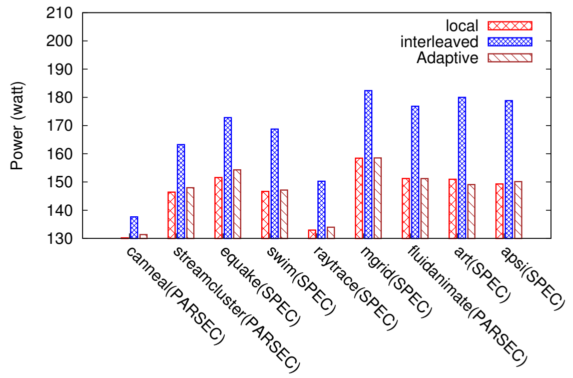
The overhead of the system is primarily concentrated in two elements, the cost of measurement and the cost of remapping the virtual cores. The measurement costs are dominated by the page table cans done by each individual core. Figure 12 shows the number of clock cycles used for the scanning process and and the remapping process for each benchmark. The highest scan cost we observed was 4.6 ms, while the highest remapping cost was 5.3 ms. To put these numbers in context, recall that these benchmarks ran for 3 to 10 minutes, with a maximum of 8 remappings. Recall that scanning is activated only then certain conditions are met, and then runs every 10 seconds. Thus, in the worst case for our benchmarks, 4.6 ms is consumed every 10 s in scanning ($< 0.05\%$ overhead). This overhead is clearly negligible.



(a) Maximizing Performance



(b) Minimizing Energy



(c) Minimizing Power

Figure 10: Performance of the adaptive system for each of the three goals. The adaptive system can dynamically and automatically select a mapping that optimizes for the goal.

7. RELATED WORK

The notion of mapping threads to cores has been studied extensively in the literature [7, 8, 6, 21], with a range of techniques proposed for online adaptation to enhance performance and save power. However, such work does not address the problem in the context of virtual machines and a NUMA architecture. The monitoring and detection schemes differ in the VMM context and a

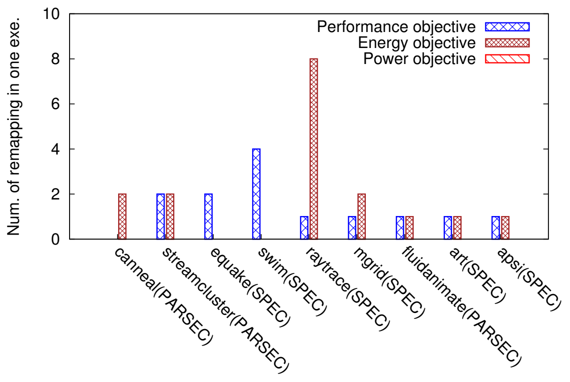


Figure 11: Number of vcore remappings during the executions of Figure 10. [3]

Benchmark	scanning (ms)	remapping (ms)
canneal(PARSEC)	1.51	5.24
streamcluster(PARSEC)	0.78	5.27
equake(SPEC)	0.82	5.25
swim(SPEC)	2.34	5.08
raytrace(PARSEC)	0.39	5.24
mgrid(SPEC)	0.61	5.27
fluidanimate(PARSEC)	0.58	5.25
art(SPEC)	1.30	5.30
apsi(SPEC)	4.61	5.27

Figure 12: Page table scanning time (average per each scan) and virtual core remapping time (moving 4 vcore threads) in milliseconds.

NUMA architecture requires that the adaptation mechanism incorporate memory locality.

Siddha et al [27] propose power-aware scheduling for Linux threads. Their scheduling policy is similar to ours in that it either statically packs threads onto a socket or core or distributes them over as many sockets or cores as possible. In contrast, our policy is dynamically adaptive and it operates on virtual cores, not threads.

Several studies investigated ways of reducing resource contention, with one of the promising approaches to have emerged recently being contention-aware scheduling [17, 24, 32]. A contention-aware scheduler identifies threads that compete for shared resources of a memory domain and places them into different domains. Most closely related to our work is that of Blagodurov et al [5], who present a contention-aware scheduler for NUMA systems that is designed to mitigate contention between applications. It provides sharing support by attempting to group threads of the same application and their memory on the same NUMA node as long as co-scheduling multiple threads of the same application does obviate a contention-aware schedule. If it needs to migrate threads to different NUMA domains, the scheduler identifies hot pages using instruction-based sampling and moves them to the new domains. Tam et al [28] discuss grouping threads of the same application that are likely to share data onto neighboring cores to minimize the costs of data sharing between them. They use the hardware PMU to track the sharing pattern between threads.

AMPS [22] is an OS scheduler for asymmetric multicore systems that supports NUMA architectures. It introduces a NUMA-aware migration policy that can allow or deny thread migration requested by the scheduler. The *resident set size* of a thread is defined and used in deciding whether or not the proposed OS schedule should be allowed to migrate thread to a different domain.

The VMware ESX hypervisor [1] supports NUMA load balancing and automatic page migration for its virtual machine (VMs). ESX Server assigns each virtual machine a home node at launch time and changes its home node periodically for load balancing. To eliminate possible remote access penalties to the old node, it migrates the hot memory pages from the original node to its new home node. Goglin et al [12] develop a memory system-aware implementation of the move_pages system call in Linux, which allows the dynamic migration of large memory areas to be significantly faster. Ibrahim et al [14] study different configurations to optimize performance in virtualized environments running on multi-socket multicore systems. It shows that optimal performance can be achieved by partitioning physical cores across multiple virtual machines and span each virtual machine across different NUMA domains.

Power and/or thermal management on multicore processor is widely discussed as limiting scalability [10, 13]. Dynamic thread scheduling on homogeneous [25] and heterogeneous [30] multicores looks promising for addressing this issue. Our work is in this vein.

8. CONCLUSIONS AND FUTURE WORK

We have demonstrated the opportunity for optimizing for performance, power, and energy presented by being able to simply choose between local and interleaved mappings of virtual cores to physical cores. The core of the paper showed how this opportunity can be leveraged in an automatic adaptive system that chooses between these two mappings based on predictions of the interactions between the workload's memory reference behavior and the mappings. We implemented and evaluated a system to do this. These two mappings represent only a tiny portion of the space of possible virtual core to physical core mappings, and we have not yet considered the mappings of guest memory to physical memory. We are currently working on formalizing a general adaptation problem that captures this space, and developing techniques for solving it.

9. REFERENCES

[1] VMware ESX Server 2 NUMA Support, White Paper. web page. http://www.vmware.com/.

[2] Alameldeen, A., and Wood, D. Ipc considered harmful for multiprocessor workloads. *IEEE Micro 26*, 4 (July–August 2006), 8–17.

[3] Aslot, V., Domeika, M., Eigenmann, R., Gaertner, G., Jones, W. B., and Parady, B. Specomp: A new benchmark suite for measuring parallel computer performance. In *Proceedings of the Workshop on OpenMP Applications and Tools* (2001).

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

[5] Blagodurov, S., Zhuravlev, S., Dashti, M., and Fedorova, A. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX)* (2011).

[6] Curtis-Maury, M., Blagojevic, F., Antonopoulos, C. D., and Nikolopoulos, D. S. Prediction-based power-performance adaptation of multithreaded scientific

[3]Note that number of remappings for power objective is always zero.

codes. *IEEE Transactions on Parallel and Distributed Systems 19*, 10 (October 2008), 1396–1410.

[7] CURTIS-MAURY, M., DZIERWA, J., ANTONOPOULOS, C. D., AND NIKOLOPOULOS, D. S. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proceedings of the 20th Annual International Conference on Supercomputing (ICS)* (2006).

[8] CURTIS-MAURY, M., SINGH, K., MCKEE, S. A., BLAGOJEVIC, F., NIKOLOPOULOS, D. S., DE SUPINSKI, B. R., AND SCHULZ, M. Identifying energy-efficient concurrency levels using machine learning. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing (CLUSTER)* (2007).

[9] DONG, M., AND ZHONG, L. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys)* (2011).

[10] ESMAEILZADEH, H., BLEM, E., ST. AMANT, R., SANKARALINGAM, K., AND BURGER, D. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)* (2011).

[11] EYERMAN, S., AND EECKHOUT, L. System-level performance metrics for multiprogram workloads. *IEEE Micro 28* (May 2008), 42–53.

[12] GOGLIN, B., AND FURMENTO, N. Enabling high-performance memory migration for multithreaded applications on linux. In *Proceedings of the 2009 IEEE Internationalon Parallel and Distributed Processing Symposium (IPDPS)* (2009).

[13] HARDAVELLAS, N., FERDMAN, M., FALSAFI, B., AND AILAMAKI, A. Toward dark silicon in servers. *IEEE Micro 31*, 4 (July–August 2011), 6–15.

[14] IBRAHIM, K. Z., HOFMEYR, S., AND IANCU, C. Characterizing the performance of parallel applications on multi-socket virtual machines. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (2011).

[15] INTEL CORPORATION. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide Part 2*, December 2011.

[16] KANSAL, A., ZHAO, F., LIU, J., KOTHARI, N., AND BHATTACHARYA, A. A. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC)* (2010).

[17] KNAUERHASE, R., BRETT, P., HOHLT, B., LI, T., AND HAHN, S. Using os observations to improve performance in multicore systems. *IEEE Micro 28* (May 2008), 54–66.

[18] LANGE, J., DINDA, P., HALE, K., AND XIA, L. An introduction to the palacios virtual machine monitor—release 1.3. Tech. Rep. NWU-EECS-11-10, Department of Electrical Engineering and Computer Science, Northwestern University, October 2011.

[19] LANGE, J., PEDRETTI, K., DINDA, P., BAE, C., BRIDGES, P., SOLTERO, P., AND MERRITT, A. Minimal-overhead virtualization of a large scale supercomputer. In *Proceedings of the 2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (March 2011).

[20] LANGE, J., PEDRETTI, K., HUDSON, T., DINDA, P., CUI, Z., XIA, L., BRIDGES, P., GOCKE, A., JACONETTE, S., LEVENHAGEN, M., AND BRIGHTWELL, R. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)* (April 2010).

[21] LI, D., NIKOLOPOULOS, D., CAMERON, K., DE SUPINSKI, B., AND SCHULZ, M. Power-aware mpi task aggregation prediction for high-end computing systems. In *Proceedings of the 2010 IEEE Parallel and Distributed Processing Symposium (IPDPS)* (April 2010).

[22] LI, T., BAUMBERGER, D., KOUFATY, D. A., AND HAHN, S. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (Supercomputing / SC)* (2007).

[23] LIBERMAN, J., AND KOCHHAR, G. *Optimal BIOS Settings for High Performance Compting with PowerEdge 11G Servers*, updated 23 august 2010 ed. Dell Product Group, July 2009.

[24] MERKEL, A., STOESS, J., AND BELLOSA, F. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)* (2010).

[25] RANGAN, K. K., WEI, G.-Y., AND BROOKS, D. Thread motion: fine-grained power management for multi-core systems. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)* (2009).

[26] RIVOIRE, S., RANGANATHAN, P., AND KOZYRAKIS, C. A comparison of high-level full-system power models. In *Proceedings of the 2008 Workshop on Hot Topics in Power-aware Computing and Systems (HotPower)* (2008).

[27] SIDDHA, S., PALLIPADI, V., AND MALLICK, A. Process scheduling challenges in the era of multi-core processors. *Intel Technology Journal 11*, 4 (November 2007).

[28] TAM, D., AZIMI, R., AND STUMM, M. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In *Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys)* (2007).

[29] UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A., MARTIN, F., ANDERSON, A., BENNETTT, S., KAGI, A., LEUNG, F., AND SMITH, L. Intel virtualization technology. *IEEE Computer* (May 2005), 48–56.

[30] WINTER, J. A., ALBONESI, D. H., AND SHOEMAKER, C. A. Scalable thread scheduling and global power management for heterogeneous many-core architectures. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2010).

[31] ZHANG, L., TIWANA, B., QIAN, Z., WANG, Z., DICK, R. P., MAO, Z. M., AND YANG, L. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Co-design and System Synthesis (CODES/ISSS)* (2010).

[32] ZHURAVLEV, S., BLAGODUROV, S., AND FEDOROVA, A. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the 15th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2010).