MAESTRO: Orchestrating Predictive Resource Management in Future Multicore Systems

Sangyeun Cho Socrates Demetriades
Computer Science Department, University of Pittsburgh
{cho,socrates}@cs.pitt.edu

Abstract

In this position paper, we make a case for a novel framework called MAESTRO *which predictively manages system resources in shared-memory parallel computing platforms built with advanced multicore processors. In such platforms, effectively coordinating the use of asymmetric shared system resources under complex program execution scenarios becomes hard. Current resource management strategies are mostly reactive and have limited awareness of an application's resource usage and asymmetry in hardware resources. For better resource management,* MAESTRO *monitors the program execution environment (hardware/OS) and application behaviors, learns useful knowledge from collected information, annotates the results of the learning to relevant program and system control structures, and makes resource management decisions such as task mapping and cache partitioning in a predictive manner.*

1. Introduction

The context of this paper is new shared-memory parallel computing platforms built with advanced multicore processors. In such platforms, effectively coordinating the use of asymmetric shared system resources under complex workload scenarios becomes hard. Most system resources in current platforms are considered homogeneous, making their management relatively simple. However, future multicore systems present significantly more asymmetry as: (1) Designers adopt asymmetry to achieve better performance, power and scalability, e.g., single-ISA heterogeneous chip multiprocessors [20], non-uniform cache architecture (NUCA) [17], switched on-chip networks [3]; (2) Process variations render processor cores *unintentionally* asymmetric in terms of their performance and power consumption [4]; (3) Imperfect resource management strategies result in unbalanced and unfair resource usages, e.g., cache contention, thermal emergencies; and (4) Growing fragility in process technology will give rise to intermittent and permanent faults while a system is operational [30]. The difficulty of tackling significant asymmetry in system resources will only grow as the amount of such resources increases.

This work was supported in part by the US NSF grants: CCF-1064976, CCF-1059283, and CCF-0702236.

The currently viable processor design and resource management practices are not particularly suitable for handling growing asymmetry in system resources, complex task mixes, and expected parallel applications in the future. For example, Frachtenberg [12] showed that the performance of a parallel application is severely affected by multiprogramming on a multicore system due to contention and the OS scheduler's lack of adequate support for parallel applications. Moreover, current OS schedulers are not aware of a program's thermal behavior or processors' heterogeneous power consumption characteristics. As such, a serious problem of existing multicore systems is that their resource management strategies *do not consider* application behaviors that are critical for informed management decisions and *are unaware* of the asymmetry of the underlying execution environment.

1.1. Impact of asymmetry, examples

How will, then, the growing asymmetry in system resources impact the performance of a large-scale multicore system, and how can an awareness of such asymmetry and application behavior help improve the system performance? As an example, consider the processor cores in a multicore system. There are at least four types of asymmetry: (1) Function: certain cores may lack native hardware support for specific operations, e.g., SIMD instructions; (2) Geometric location: depending on "where" a task runs, it experiences disparate latencies to caches and memory controllers; Mismatching a task may result in performance degradation; (3) Performance/power: not all cores have the same performance/power characteristics due to design time decisions or process variations; and (4) Core state: cores may be in different states (e.g., temperature) at the time of task mapping.

Figure 1(a) shows how variations in maximum core speed affect the outcome of task scheduling in a latency-oriented system. We compare two conventional task mapping schemes, "*Random*" and "*Load Balancing*," with a variation-aware scheme, "*Aware*." *Random* picks a core for a task randomly to balance core usage. *Load Balancing* chooses a core with the shortest task queue for new task mapping. *Aware* uses a policy where the fast cores are used before slow cores and the task queues are balanced with the knowledge of a task's execution time. *Aware* is shown to outperform both *Random* and *Load*

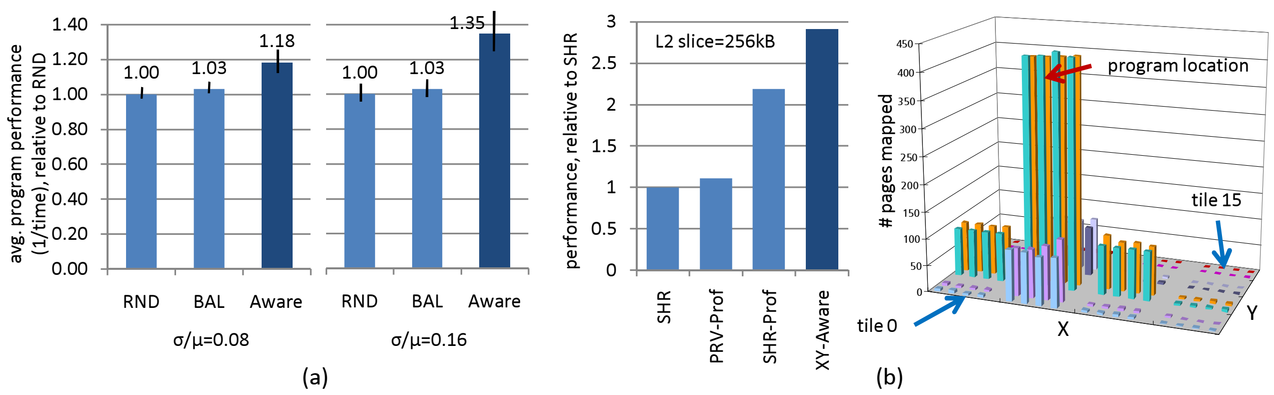


Figure 1. (a) Random (RND), Load Balancing (BAL), and Variation-Aware (Aware) task mapping schemes. Core speed variations were injected into a 32-core processor using Gaussian distribution with standard deviation σ and average μ. We ran a multiprogrammed workload where jobs arrive to maintain a 4% system load. The right graph shows the same test with higher variation. (b) Shared (SHR), Private with Profiling (PRV-Prof), Shared with Profiling (SHR-Prof), and XY Location-Aware (XY-Aware) data mapping schemes [15]. We run `mcf` (SPEC2k) on a 16-core machine using a 2D mesh network. The right figure shows the distribution of page mappings made by XY-Aware.

Balancing. It is also shown that performance difference grows as the speed variation among cores increases.

Last-level cache memory is an important shared resource in a multicore system. Figure 1(b) illustrates how mapping pages to be physically closer to the core using them can improve performance over conventional schemes. Shared ("SHR") distributes cache blocks to all cache slices [29]. Private with Profiling ("PRV-Prof") always maps cache blocks to the same core as the program, enhanced with profile-based page coloring. Shared with Profiling ("SHR-Prof") is same as SHR but is enhanced with page coloring. A location aware scheme ("XY-Aware") has knowledge of the network and cache organization, as well as an application's page access behavior [15]. It places pages on or near the tile running the application to reduce cache latency and cache misses.

We presented only two motivating examples; however, it is not hard to find other system management scenarios that will benefit from the knowledge of applications and execution environment. We will discuss more examples in Section 2.2 and 3.

1.2. Our approach and vision

We envision *a multicore system where the complex task of coordinating resource usage among multiple applications is dealt with predictively based on application and system knowledge; the knowledge is produced and maintained automatically.* We propose a framework to enable such a system and call it MAESTRO.

In MAESTRO, (1) Program execution environment (hardware and the OS) and application behaviors are opportunistically monitored; (2) Useful knowledge is learned from the collected information; the results of the learning are then annotated to relevant program and system control structures; and (3) System resource management decisions such as task scheduling and cache partitioning are made in a predictive manner by exploiting the accumulated knowledge. MAESTRO sits in the conventional system resource management layer (OS and virtual machine monitor (VMM)) and orchestrates the actions involved in the process. Another property of MAESTRO is its transparency—it requires little-to-no human intervention and will not interfere with the user's regular computer usage. For instance, when the system becomes idle (e.g., the user is at lunch or off duty), MAESTRO may initiate "proactive" profiling by test-running an application with different input sets generated automatically.

1.3. Paper organization

In what follows, we will first describe our proposed framework in Section 2. In this section, after giving an overview of how MAESTRO operates, we will briefly discuss example problems and detail our ongoing research tasks that must be undertaken before a practical MAESTRO system can be realized. To put our proposal in perspective, Section 3 presents a case study that explores dynamically adjusting voltage and frequency of a processor's on-chip network. After summarizing related work in Section 4, conclusions will be drawn in Section 5.

2. MAESTRO

2.1. Overview

MAESTRO's key components are *Application Profiler*, *Environment Profiler*, *Profile Analyzer*, *Annotations*, *Resource Manager*, and *Conductor* as depicted in Figure 2. Important functional requirements of each MAESTRO component are as follows. Application Profiler gathers detailed information about an application's run-time behavior. Data are collected with three different levels:

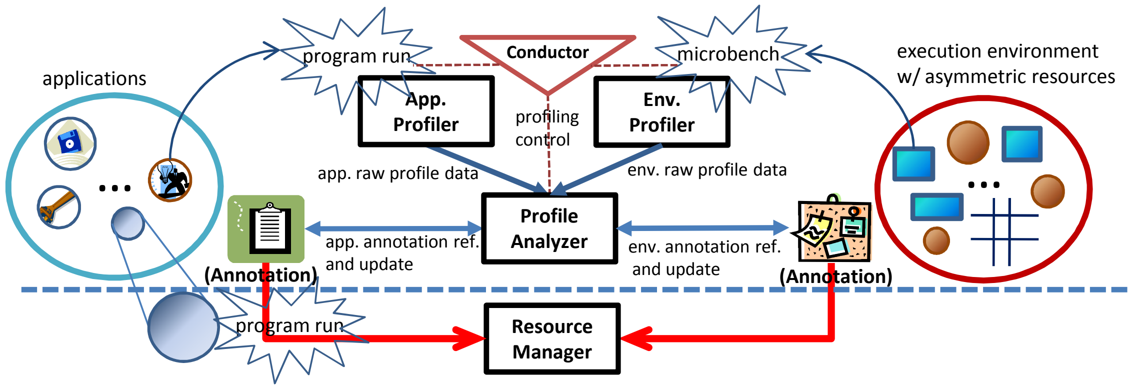


Figure 2. Conceptual diagram of MAESTRO components (in solid boxes) and their interactions: Application Profiler, Environment Profiler, Profile Analyzer, Annotations, Resource Manager, and Conductor. Solid arrows represent information flow and dotted arrows show control flow. Actions above the horizontal dotted line are for profiling.

the whole program, program phases, and individual instructions. Application Profiler should operate with minimal intrusiveness to the application being profiled by taking advantage of hardware-based instrumentation facilities such as performance counters and other idle cores.

Environment Profiler characterizes the execution environment by observing the behavior of the system using both system-level and hardware-level metrics. Data are collected with regard to specific system structures such as processor core, last-level shared cache and memory controller. Profile data are obtained from running and monitoring real applications or specially designed microbenchmarks. As with Application Profiler, we want to minimize the performance impact by utilizing idle CPU cycles and hardware monitoring facilities.

Profile Analyzer analyzes raw profile data produced by Application Profiler and Environment Profiler. The outcome of Profile Analyzer is useful knowledge of an application's resource usage and the execution environment's characteristics. Profile Analyzer uses machine learning techniques such as clustering and supervised learning to efficiently handle potentially large multi-dimensional data. Machine learning is also used when we have partial, incomplete information. Profile Analyzer works off-line and does not interfere with normal system operations.

Annotations collectively form the knowledge of a system learned over time. Conceptually, there are annotations for each application and annotations for each allocatable resource. Annotation data structures must support efficient retrieval of information.

Resource Manager makes resource management decisions using the knowledge obtained from Annotations. For example, it could minimize system energy consumption subject to quality-of-service (QoS) requirements.

Conductor controls the actions of other agents. It can initiate profiling applications or the use of a specific resource. Conductor needs to track day-to-day and user-to-user system usage patterns and recognize idle CPU cycles to automatically initiate proactive profiling. It may also function as an interface to the user, setting abstract goals for Resource Manager like optimizing energy, performance, maintaining target QoS; the degree of profiling for a program; or disk space for profiling.

Among the MAESTRO components, Application and Environment Profiler will benefit greatly from special architectural support to track run-time events and processor states. Other agents do not require special support.

2.2. Example resource management problems

Let us present three example resource management problems to highlight how the knowledge of individual programs and an execution environment can help.

Intelligent initial task mapping. In a conventional system, a new task is mapped to a processor based on a simple policy, e.g., random, round-robin, or where task queue length is the shortest. Conventional schemes do not perform well in the presence of core speed variations (as illustrated in Figure 1(a)) or when cores exhibit functional discrepancies [21]. With the knowledge of applications and the characteristics of available processor cores, one will be able to define an objective function that gives a "score" for each possible task-core mapping, with which we can determine the best target core. The objective function takes the annotations of a core and a task to compute the corresponding score. The annotation for a task may include a vector comprised of the usage count of available compute resources (e.g., FP multiplier, MMX unit). The annotation for a core can feature a vector comprised of the cost of each compute resource usage. In this case, the score could be the inner product of the two vectors. Likewise, we can define a scoring function that incorporates power and performance ratings of individual cores.

Last-level cache management. The two key problems in the last-level shared cache management are overcoming non-uniform cache latencies and controlling capacity allocation and inter-task contention. A MAESTRO system could learn an application's input-dependent working set and spatial/temporal cache access behavior, and predictively coordinate data mappings and data migration when the application runs. The collected information will be useful also for achieving intelligent initial task mapping and resource provisioning on clustered cache architectures and multi-socket systems.

Power and energy management. We can utilize cores' different power consumption rates; in this case, low-power cores will be selected to run a task first as long as the task's QoS is met. Benefits of such per-core power awareness will be high when the processor cores see large variations and the workload is heterogeneous. Application awareness can also help. For instance, if we have an accurate prediction of an application's run time and phase behavior, we can utilize DVFS in a predictive manner (by pre-computing per-core voltages and frequencies) to save energy while still meeting its performance goal. Previous DVFS work for multithreaded workloads [7] unrealistically assume a priori knowledge of the workload; with the prediction capabilities of MAESTRO, the ideas of the prior work can have sizable practical benefits.

2.3. Research tasks

2.3.1. Characterizing resource asymmetry

Our first research task is to characterize the impact of resource asymmetry on programs' run-time behavior and to extract key architecture and system control parameters for each management problem. We can then formulate our target resource management problems using the parameters we will have identified. The types of interesting asymmetry are processor performance and power, on-chip cache access latency, and off-chip memory access latency.

An important issue that has not been studied thoroughly is the *dependency* of one parameter on another. For example, when a program runs on a fast core, it will generate L2 cache accesses at a faster rate than on a slow core. Hence, one will have to allocate more memory bandwidth to the program to fully expose the benefit of the fast core. On the other hand, if we allocate a large L2 cache capacity to a program, memory bandwidth consumed by the program will be reduced with fewer L2 misses, and thus, allocating more cache capacity and memory bandwidth together to favor a program may lead to wasting memory bandwidth.

2.3.2. Learning an application

The two important questions for this task are "what do we learn from an application?" and "how do we learn?" The former depends on what knowledge is useful for Resource Manager and the latter on the accuracy and efficiency of the learning methods under consideration.

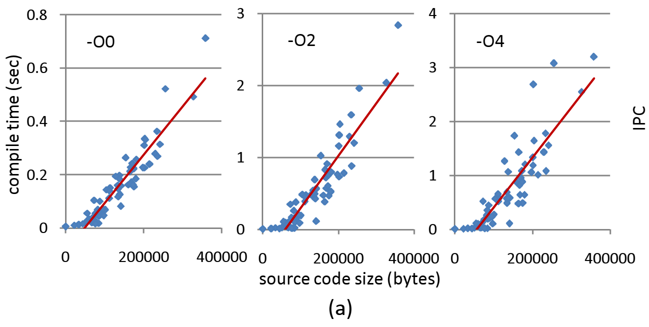
We take into account three levels for learning and associating information with: the whole program, program phases, and individual instructions. Whole program knowledge includes execution time and working set as a function of program input. A program's working set, typically represented as a miss rate curve against memory capacity, will be essential for predicting memory system performance and intelligent cache partitioning. In addition, a program might be labeled with whether it is a real-time program (and its requirements) or a multithreaded application (and its communication behavior). Per-phase knowledge can include phase boundaries, resource needs, and inter-thread communication patterns. Instruction-oriented knowledge provides hints about functional unit usages and how trends of fine-grained architectural events like cache misses and synchronization actions change at specific program points.

Let us consider predicting a program's execution time. Existing estimation methods consider program execution history such as job submission time, job owner, and run times of similar programs [13]. In principle, however, a program's resource-unconstrained run time is determined by the input to the program. We propose to directly learn a program's execution time function—a function of program input—over a series of profiling during program runs either mandated by the system user or orchestrated by Conductor with carefully synthesized program input sets. The complexity of this task depends on the number of inputs, the number of distinct values each input can assume, and the algorithmic nature of the program. While the task of learning a program execution time function sounds daunting, our preliminary examination of popular benchmarks suggests that program execution times depend on a relatively small number of (typically one or two) inputs independently and the functions tend to be quite simple, e.g., linear. As an example, Figure 3(a) depicts the trend of `gcc`'s run time against input source code size. It is shown the run time is linear to source code size despite higher optimization levels add variations. We expect that in practice, approximate predictions are helpful to many resource allocation decisions.

For the phase-oriented profiling, resource usage of a program can be thought of as a time-varying function $R(t)$. Given K resource types and an unconstrained program run time of T, the combined resource usage is:

$$\vec{R}(t) = (R_1(t), R_2(t), ..., R_K(t))_{t=0\sim T} \qquad (1)$$

If a program run is a series of phases and resource usage is relatively unchanging within each phase, resource usage of a phase ($\vec{R}_i$) and of a program ($\vec{R}$) can be written:



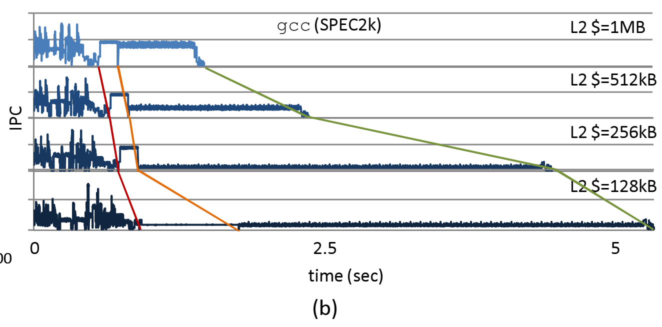


Figure 3. (a) Run time of `gcc` (v3.4.6) as a function of source file size, measured on a Linux box. Input files are 65 C files of `gcc` in SPEC2k. (b) Phase lengths change differently as L2 cache size is varied.

$$\vec{R}_i = (r_{i1}, r_{i2}, ..., r_{iK}) \quad (2)$$
$$\vec{R} = < \vec{R}_1 \mid \vec{R}_2 \mid ... \mid \vec{R}_M > \quad (3)$$

where i is a phase number and M is the number of phases. When actual resource allocation is described by $\vec{A} =< \vec{A}_1 \mid \vec{A}_2 \mid ... \mid \vec{A}_M >$ where $\vec{A}_i$ and $\vec{A}$ are similarly defined as (2) and (3), the execution time of the application (i.e., the sum of all phase execution times) will be determined by the mismatch of $\vec{R}$ and $\vec{A}$. This observation forms the basis for Application Profiler to perform phase-oriented profiling. By monitoring an application multiple (say N) times using $\vec{A}_i$ ($i = 1...N$), one will be able to effectively profile resource sensitivity of all observed phases. Figure 3(b) shows how phase lengths change differently as L2 cache size is changed from 1MB down to 128kB. While program phase detection algorithms have been researched extensively (e.g., [25–28]), low-cost dynamic phase detection methods using multiple observations and OS-visible resource usage oriented characterization have not been studied well, especially for anticipated multithreaded workloads. Hence, we will explore new frameworks and algorithms to efficiently detect and characterize phases in multithreaded applications.

Instruction-oriented profiling is used to characterize the usage of the processor's functional units and to detect fine-grained program behaviors that are not dependent on program inputs or the amount of resources allocated. For instance, if a particular store is always followed by a load from another thread, indicating a producer/consumer instruction, we can preemptively send the data to the consumer thread's core to avoid a load miss in that core. Another example is the functional unit usage by instructions.

Application Profiler has two operation modes, *passive* and *proactive*. Passive profiling is performed when a user launches an application, enough resources are available for profiling, and Conductor determines profiling is actually needed (e.g., the quality of current annotations is low). Proactive profiling is triggered by Conductor, which opportunistically exploits idle CPU cycles. Under proactive profiling, an application will be run multiple times on deliberately different input sets and on differently configured system resources.

2.3.3. Learning an execution environment

This research task will study how to profile an execution environment from the viewpoint of intelligent resource management. First of all, we will learn asymmetry present in system resources. Much of the design-time information can be made conveniently available in a MAESTRO system. For example, at boot time, the OS can retrieve the processor type/version and look up in its pre-compiled database desired hardware information. Further, to consider "unintentional asymmetry," we must dynamically characterize processor cores' native support for specific operations, non-uniform cache/memory access latency, performance, and power characteristics.

Recently, there is an increased amount of low-level fault and degradation information made available to the system. For example, common NAND flash devices come with initial "bad block markings" from the manufacturer [24]. IBM Power series processors [29] implement built-in self test logic that checks cache memories after power on. Researchers have even developed special area-efficient sensors to directly measure circuit performance degradation [16]. Our focus in this task will be to develop architectural and software strategies that collaborate with such circuit-level support.

For profiling, data obtained during regular user-initiated application runs (i.e., passive profiling) will be useful. However, we expect profile data obtained from running micro-benchmarks (i.e., proactive profiling) will be more accurate. For example, a micro-benchmark can exercise a specific processor core at a parameterized rate and measure how quickly the core gets hot.

Finally, execution environment profiling is not a frequent activity and new profiling is invoked only once in a while, e.g., every month, to adapt to any changes in system environments.

2.3.4. Architectural support

While today's microprocessors provide hardware performance counters, MAESTRO will benefit from new efficient instrumentation facilities that can capture more complex and informative events. Specifically, the following four types of monitoring support appear especially helpful: (1) fine-grained processor functional unit usage counters, (2) specific instruction tracking, (3) power meter, and (4) monitoring data sampling, analysis, capturing and storing support.

The fine-grained processor functional unit usage counters can quickly characterize an application's resource usage behavior at the instruction level. The specific instruction tracking support will help characterize an application's instruction- and phase-level behaviors by identifying specific instructions present in the application, such as synchronization and transaction primitives. The power meter, together with other performance counters, will allow us to characterize the compute cores in terms of their power and performance. With the modern processor's capability to turn off individual cores, it is sufficient to have a coarse-grained (off-chip) power meter that reports the total chip power. By selectively turning on/off cores and running microbenchmarks, one would be able to characterize individual cores. To reduce the amount of data to collect, flexible sampling support is desirable. Moreover, rather than obtaining simple event counts, one may desire more sophisticated information such as when a counter matches a pre-defined value or when a large change in event frequency is detected. Lastly, the captured events could be logged automatically (to a designated memory buffer or persistent storage).

3. A Case Study

This case study evaluates the effectiveness of a predictive energy and performance management scheme by a MAESTRO system. In this system, application phases are observed at the "epoch" granularity, which is a time interval between global synchronization points [8]. Management decisions are taken predictively based on epoch profile information that is collected either during profile runs or at actual run time. The scheme aims to optimize the energy and performance trade-off in a chip multiprocessor using the DVFS capability of the NoC. We consider a simple NoC DVFS configuration in which all the routers of the network comply to the same voltage/frequency setting at a given time.

We note that the purpose of this case study is not to propose a complete solution for the energy/performance management of the NoC; rather, we aim to demonstrate the applicability of our proposed approach in the context of dynamic adaptation.

CAPTION	FREQUENCY	VOLTAGE
$f_{100\%}$	3 GHz	0.8 V
$f_{75\%}$	2.25 GHz	0.65 V
$f_{50\%}$	1.5 GHz	0.5 V
$f_{25\%}$	0.75 GHz	0.35V

Table 1. Frequency/voltage levels.

3.1. Epoch-based adaptation

We have two different implementations of epoch-based adaptation. In the first one, off-line profiling determines the frequency/voltage setting best suited for each epoch of the application ("static scheme"). We perform a profile run for each NoC voltage/frequency setting to record the execution time (D_f) and energy consumed (E_f) by each epoch (the sum of all its dynamic instances). Then, we pick, for each epoch, the frequency/voltage level f_x for which $E_{f_x} \times D_{f_x}$ is minimized.

In the second implementation, the epoch table and the decision signatures are determined dynamically at run time ("dynamic scheme"). A search for the best frequency level is triggered at the first instance of each epoch. During this period, all possible NoC voltage/frequency settings are examined by switching to a different frequency on a fixed time interval basis within the epoch instance. Because the energy and performance measurements for each frequency have to be taken from different time intervals, we measure energy per instruction (EPI) and cycles per instruction (CPI). Thus, the best frequency f_x for each epoch becomes the one that minimizes $CPI_{f_x} \times EPI_{f_x}$.

Clearly, the static scheme is more effective than the dynamic scheme since optimal decisions will be readily available at run time. However, the dynamic scheme is more general and is applicable to a wider range of environments. We note that even in dynamic environments, users often tend to execute a limited number of applications frequently, possibly solving similar problems repeatedly. MAESTRO's capability to capture the knowledge of the applications will be effective in both cases.

3.2. Experimental setup

The experiments are performed using a detailed multicore processor simulator on Simics [23]. In our simulator, the NoC models a wormhole-switched network with deterministic X-Y routing and ACK/NACK flow control. Data packets consist of six 128-bit flits and control messages one flit. Each router models a two-stage router pipeline and has five physical channels (PCs) and two Virtual Channels (VCs) multiplexed on each PC.

For NoC DVFS, we assume four possible clock frequency and voltage levels, as shown in Table 1. $f_{100\%}$ represents the maximum operating frequency of the NoC, which we consider as the baseline frequency (no energy savings). DVFS policies are triggered during epoch

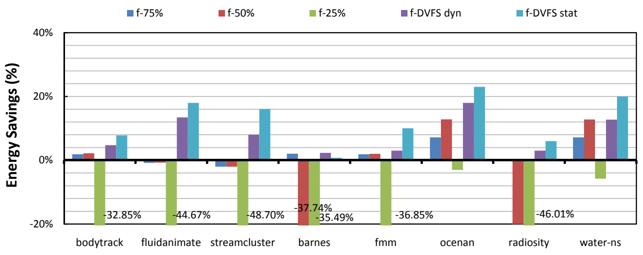


Figure 4. Energy savings (relative to the baseline fixed scheme $f_{100\%}$): Reducing the frequency/voltage of NoC can result in overall energy savings.

transitions; assuming on-chip voltage regulators, we account 100 cycles for switching overhead. For the dynamic implementation, we use 100k sampling intervals and thus the per-epoch monitoring period lasts at least 500K cycles—100k for warm-up + 100k (one sample) per frequency level. During this period, we keep high voltage and we switch only frequency, therefore we account zero switching overhead, while the energy consumption for the corresponding frequency level is estimated.

Power consumption is modeled with dynamic, leakage, and background components. The background power represents the cores and the rest of the system and is not in the same clock domain. The dynamic and leakage power numbers are extracted based on the assumption that at 1 GHz, the leakage power consumption is twice that of the dynamic power, which is consistent with estimates from Kim et al. [18]. The background power is constant and is computed assuming that a loaded NoC at 1 GHz consumes 30% of the total system power [19].

3.3. Results

Figure 4 and 5 each shows the total energy savings and execution slowdowns of the studied epoch-based adaptation schemes, as well as non-adaptive fixed frequency schemes. Results are normalized to the baseline case, where the NoC operates at $f_{100\%}$. As the NoC frequency decreases, the power consumed in the NoC is reduced and the overall on-chip energy savings is expected to grow. On the other hand, more time is needed for the application to complete execution, which makes it unclear whether the power benefits gained in the NoC will result in energy savings during the whole execution. For example, the results show that when the NoC operates at $f_{25\%}$, the slowdown is large enough to adverse the power savings into significant energy loss. The same also happens in some $f_{50\%}$ cases. All other cases show energy savings at the cost of performance. Note that for all benchmarks, the epoch-based adaptive schemes show energy reductions for the least performance degradation compared with the fixed schemes.

Since our optimization target is to hit the en-

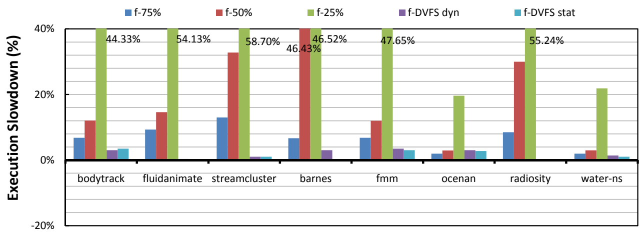


Figure 5. Execution slowdown (relative to the baseline $f_{100\%}$): Reducing the NoC frequency can lower performance.

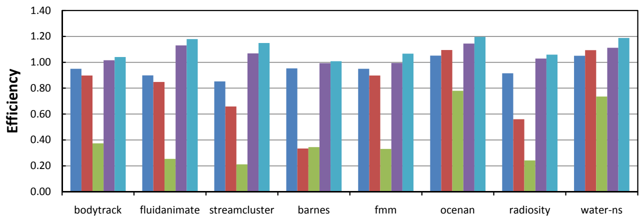


Figure 6. Efficiency: The efficiency is the product of the energy savings and performance degradation, normalized to the baseline $f_{100\%}$.

ergy/performance trade-off, we show in Figure 6 the product between the energy savings and the execution slowdown (always as a ratio to the $f_{100\%}$). The results illustrate the effectiveness of both static and dynamic schemes, in adapting the system into a more efficient state. The static scheme represents a bound on what the dynamic scheme can achieve with the current experimental setting. We find that the results with the dynamic scheme follow closely and consistently those of the static scheme, indicating the strength of the proposed approach in capturing the changes in program behavior. In barnes and radiosity where a single epoch is dominating the execution, the efficiency is directly affected by the frequency applied to that specific epoch. As the results show, both schemes reach the same decision and successfully pick the optimal frequency.

In summary, our case study strongly suggests that learning an application to optimize energy-performance efficiency can achieve sizable benefits (up to 20% in our study). The learning process was quite simple, given proper support to partition the program execution into meaningful intervals (epochs in our study). We examined and showed the promise of relatively short-range predictive adaptation (dynamic scheme, within an execution) and long-range adaptation (static scheme, across executions) that are enabled by a MAESTRO system. The asymmetry we addressed in this case study includes: different execution time contributions from different program intervals, different, non-linear sensitivity of program intervals to performance/power characteristics of the NoC, and the non-linear performance/power behavior of the underlying architecture when the NoC DVFS is applied.

4. Related Work

Due to space limitations, this section focuses only on recent feedback-directed automatic program optimization techniques and their frameworks. Traditionally, program profiling has provided detailed information on a program's dynamic behavior to a static compiler for informed code optimizations. For example, code scheduling structures such as traces and superblocks are formed with path profiling information [10, 14]. Classic code optimizations such as function inlining, dead code elimination, and loop invariant removal become more effective with profiling [6]. Dynamic code optimization frameworks incorporate monitoring and analysis of program behavior seamlessly with code optimizations at run time [1, 2, 22].

Recently, Ding et al. [9] presented a profile-driven optimization framework where application profile is used to detect phases, then the profile and phase change information is exploited to optimize application performance. Their suggested optimizations are array regrouping and structure splitting, I/O prefetching and multitasking control, adaptive memory management (e.g., garbage collection), and software-hardware cooperative memory disambiguation. They further suggested that a program's phase behavior can be used to load-balance a multithreaded multicore processor, e.g., as in [11]; this work however focuses only on mapping already co-scheduled threads to available hardware contexts. However, much of their work focuses on improving a single program with the knowledge of pre-detected phases.

Another closely related project to our research is IBM's continuous program optimization (CPO) framework [5, 31]. Like MAESTRO, it uses comprehensive system monitoring (called performance- and environment-monitoring or PEM) to learn the behavior of a target application, and statically or dynamically improve the application and its execution environment (e.g., JVM). While CPO monitors the execution environment, their goal has been to improve a single application. Their execution environment is rather abstract, e.g., object pool.

5. Conclusions

This paper advocated a self-adaptive multicore system framework called MAESTRO. With the growing asymmetry in system resources and the anticipated (heterogeneous) multithreaded workload, we argue that intelligent, predictive resource management will remain increasingly important. To enable predictive resource management, MAESTRO coordinates learning processes for applications and the execution environment that they will run on. We have identified and described the key research tasks necessary for building the overall framework in this paper. Finally, we design and present a case study that highlights the effectiveness of a predictive power-performance optimization strategy in the context of a multicore system.

References

[1] S. V. Adve et al. "Changing Interaction of Compiler and Architecture," *IEEE Computer*, Dec. 1997.

[2] V. Bala et al. "Dynamo: A Transparent Dynamic Optimization System," *PLDI*, June 2000.

[3] L. Benini and G. De Micheli. "Networks on chips: a new SoC paradigm," *IEEE Computer*, Jan. 2002.

[4] S. Borkar. "Microarchitecture and Design Challenges for Gigascale Integration," keynote speech at *MICRO*, Dec. 2004.

[5] C. Cascaval et al. "Performance and environment monitoring for continuous program optimization," *IBM J. Res. & Dev.*, Mar./May 2006.

[6] P. P. Chang et al. "Using Profile Information to Assist Classic Code Optimizations," *SPE*, Dec. 1991.

[7] S. Cho and R. Melhem. "On the Interplay of Parallelization, Program Performance and Energy Consumption," *TPDS*, Mar. 2010.

[8] S. Demetriades and S. Cho. "BarrierWatch: Characterizing Multithreaded Workloads across and within Program-Defined Epochs," *Computing Frontiers*, May 2011.

[9] C. Ding et al. "Program Phase Detection and Exploitation," *IPDPS*, Apr. 2006.

[10] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*, MIT Press, Apr. 1986.

[11] A. El-Moursy et al. "Compatible Phase Co-Scheduling on a CMP of Multi-Threaded Processors," *IPDPS*, Apr. 2006.

[12] E. Frachtenberg and Y. Etsion. "Hardware Parallelism: Are Operating Systems Ready?" *WIOSCA*, June 2006.

[13] R. Gibbons. "A Historical Application Profiler for Use by Parallel Schedulers," *Job Scheduling Strategies for Parallel Processing*, Eds. D. G. Feitelson and L. Rudolph. 1997.

[14] W.-M Hwu et al. "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *JSC*, June 1993.

[15] L. Jin and S. Cho. "Taming Single-Thread Program Performance on Many Distributed On-Chip L2 Caches," *ICPP*, Sep. 2008.

[16] E. Karl et al. "Compact In-Situ Sensors for Monitoring Negative-Bias Temperature-Instability Effect and Oxide Degradation," *ISSCC*, Feb. 2008.

[17] C. Kim et al. "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches," *ASPLOS*, Oct. 2002.

[18] N. S. Kim et al. "Leakage current: Moore's law meets static power," *IEEE Computer*, 2003.

[19] J. S. Kim et al. "Energy characterization of a tiled arch. processor with on-chip networks," *ISLPED*, 2003.

[20] R. Kumar et al. "Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction," *MICRO*, Dec. 2003.

[21] T. Li et al. "Operating System Support for Shared-ISA Asymmetric Multi-core Architectures," *WIOSCA*, June 2008.

[22] C.-K. Luk et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *PLDI*, June 2005.

[23] P. S. Magnusson et al. "Simics: A Full System Simulation Platform," *IEEE Computer*, Feb. 2002.

[24] Micron. *Design and Use Considerations for NAND Flash Memory*, TN-29-17, 2006.

[25] P. Nagpurkar et al. "Online Phase Detection Algorithms," *CGO*, Mar. 2006.

[26] E. Perelman et al. "Detecting Phases in Parallel Applications on Shared Memory Architectures," *IPDPS*, Apr. 2006.

[27] T. Sherwood et al. "Automatically Characterizing Large Scale Program Behavior," *ASPLOS*, Oct. 2002.

[28] T. Sherwood et al. "Phase Tracking and Prediction," *ISCA*, 2003.

[29] B. Sinharoy et al. "POWER5 system microarchitecture," *IBM J. Res. & Dev.*, July/Sep. 2005.

[30] J. Srinivasan et al. "The Impact of Technology Scaling on Lifetime Reliability," *DSN*, June 2004.

[31] R. W. Wisniewski et al. "Performance and Environment Monitoring for Whole-System Characterization and Optimization," *PAC2*, Oct. 2004.