Threads vs. Caches: Modeling the Behavior of Parallel Workloads

Zvika Guz[1], Oved Itzhak[1], Idit Keidar[1], Avinoam Kolodny[1], Avi Mendelson[2], and Uri C. Weiser[1]
[1]*EE department, Technion, Israel* [2]*Microsoft Corporation*
[1]*{zguz|ovedi@tx, idish|kolodny|uri.weiser@ee}.technion.ac.il* [2]*avim@microsoft.com*

***Abstract* — A new generation of high-performance engines now combine graphics-oriented parallel processors with a cache architecture. In order to meet this new trend, new highly-parallel workloads are being developed. However, it is often difficult to predict how a given application would perform on a given architecture.**

This paper provides a new model capturing the behavior of such parallel workloads on different multi-core architectures. Specifically, we provide a simple analytical model, which, for a given application, describes its performance and power as a function of the number of threads it runs in parallel, on a range of architectures. We use our model (backed by simulations) to study both synthetic workloads and real ones from the PARSEC suite. Our findings recognize distinctly different behavior patterns for different application families and architectures.

I. INTRODUCTION

Nowadays, *high-performance engines*– GPUs and similar accelerators– are becoming increasingly popular. Such engines serve the mounting computation needs of high-throughput and graphics-processing applications. Meanwhile, the body of applications targeted for such throughput-oriented machines continues to enlarge: the term GPGPU [1] reflects a broadening of the focus to include not only graphics, but also a wide range of highly-parallel applications.

As memory access is a principal bottleneck in current-day computer architectures [2], a key enabler for high performance is masking the memory overhead. Today's high-performance engines employ two design principles to overcome memory related issues: The first is based on a cache architecture that takes advantage of locality of references to memory. Intel's Larrabee [3] is a prominent example of this approach. The second approach uses aggressive multithreading so that whenever a thread is stalled, waiting for data, the system can efficiently switch to execute another thread. This approach is heavily used in current graphics processing engines such as Nvidia's GT200 [4] and AMD/ATI's Radeon R700 [5], which manage thousands of in-flight threads concurrently. Moreover, we now see the emergence of systems, like Nvidia's Fermi [6], that employ both approaches by combining large caches with numerous in-flight threads.

Since such a combination of two very different approaches is used to overcome the memory bottleneck, performance prediction becomes non-intuitive and challenging. The extent to which an application will benefit from either approach depends on many architecture and workload parameters. Moreover, the relative impact of caching compared to multi-threading changes as the number of threads scales up. This complex behavior, in turn, poses a challenge for architecture designers, who need to allocate the limited on-die resources to cores, thread contexts, and caches. Finally, given a diversity of already available high-performance architectures, there is the question of which is the best fit for a given workload.

This paper addresses these challenges by developing a simple, high-level, closed-form model that captures both the architecture and the application characteristics (see Section III). The modeled machine uses a parameterized combination of both mechanisms for memory latency masking, and can thus capture a range of machines, rendering the comparison between them meaningful. The workload model, in turn, captures the salient properties of the program, which allows one to predict which architecture is most beneficial for it. All the parameters— capturing both architecture and workload— can be used as "knobs" for studying a wide range of scenarios, in order to comprehend the interplay among multiple parameters in a clean, qualitative way. The model thus serves as a vehicle to derive intuitions.

In Section IV, we study how different properties of an application affect performance and power. We identify three families of workloads with distinct behavior patterns: While some workloads have a clear affinity towards either caching or multi-threading, others can benefit from both. Moreover, some workloads exhibit an unintuitive "valley" between the cache efficiency zone and the thread efficiency zone, where performance takes a dip.

In Section V we back our analytical model by simulations. Our results indicate that the simple, closed-form model of Section III can, in most cases, predict dynamic behavior, and can thus be used to select the most efficient hardware structure for executing a given program. Whereas Section IV concentrates on synthetic workloads, Section V studies workloads from the PARSEC benchmark suite [7], and shows that the three distinct behaviors observed in Section IV are indeed present in real workloads.

To summarize, our contributions are as follows:

- We present a *simple* closed-form model for systematically reasoning about *complex* phenomena; the model captures the behavior of parallel workloads on high performance engines that employ any combination of caching and aggressive multi-threading.
- We conduct a *qualitative* study of the inherent tradeoffs between the two approaches for memory access masking, and their sensitivity to a range of parameters. Our study yields non-intuitive observations regarding the impact of architectural choices and workloads characteristics on performance and power.

978-1-4244-8935-0/10/$26.00 ©2010 IEEE

- We validate our model via simulation of real workloads.

Finally, we believe that our model can direct further research on ways to address the memory wall problem in high-performance engines.

II. Related Work

Though there are many existing analytical models for processors' performance, they mostly concentrate on a single family of processors (either multi-core, multi-thread, GPU, etc.), and thus on one of the two paradigms – either caching or multithreading. For example, previous analytical models that analyzed caching in multi-threaded processors [8][9][10] considered only a small thread count, and hence are not applicable for machines that manage thousands of in-flight threads. Previous models for GPU machines [11][12], have not considered large caches, and hence are not applicable for machines where caches are a key factor in determining performance. To the best of our knowledge, our model is the first to specifically target the interplay of the two paradigms and is the first to model both via a single, unified model that enables a clean comparison across the design space of the new generation of high performance engines.

In a preliminary work [13] we presented the core of the analytical model and the existence of a valley-like behavior. In Section III, we extend this model with power equations and account for performance-power tradeoffs. Unlike [13], we study real benchmarks, and refine our initial observations by identifying three distinct types of workload behaviors with different performance curves. Additionally, unlike [13], we use simulations to validate the theoretical model.

Previous characterizations of the PARSEC benchmark suite [7][14][15] concentrate on machines with significantly fewer cores than we do, and parallelization only up to 32 threads. We push multi-threading as well as the number of computation units to the hundreds.

III. The Analytical Model

In order to study the basic tradeoffs of caching and multithreading over the range of high performance engines and applications, we use a high-level, abstracted model that can capture both mechanisms. This abstraction enables us to derive specific instances for different machines from the same unified framework in a way that renders the comparison meaningful. To enable elementary reasoning of the basic tradeoffs, we purposely use a simple, first-order model. Indeed, the model can be augmented to account for various additional effects, but this should come second to the basic tradeoff of caches vs. threads captured in this paper.

While different architectures may differ in their programming model, we do not consider programming issues here. Rather, we assume that the same applications can be mapped to different engines across the range; frameworks like OpenCL [16] and Ocelot [17] are expected to allow for such cross-platform mappings. The different models, however, are commonly described using different terminologies, which can be confusing. Our terminology follows the one used in multi-threaded programming models like CUDA [18], where a *thread* is a basic execution stream that processes a single data element. A *processing element* (*PE*) is a processing unit that processes a single such light-weight thread at a time; CUDA also uses the term *Streaming Processor* (*SP*) for a PE. In programming models like *Larrabee Native* [3], each core executes a SIMD instruction that processes several (e.g., 16) data elements at the same time. Thus, in our terminology, a Larrabbe core is composed of 16 PEs, which can execute 16 threads at a time. (Note that our notion of threads is different from traditional operating systems threads; such light-weight threads are called *strands* in Larrabee Native.)

A. Hardware and Workload Model

Our abstracted machine includes an array of N_{PE} processing elements and a large on-chip shared cache of size $S_{\$}$. For simplicity, we only model the shared cache (L2/L3), and consider local L1 caches, if they exist, to be part of the processing element. In addition, the machine includes a register file for storing the contexts of up to N_{max} in-flight threads; we assume that this is the maximum number of threads running concurrently. We consider simple, in-order PEs, for which the average number of cycles required to execute an instruction is CPI_{exe} (assuming a perfect, zero-latency memory system). We assume that the machine is symmetric; hence all PEs run at the same frequency *f*. The on-chip cache latency is $t_{\$}$, while the off-chip memory can be accessed at a latency of t_m cycles, and a bandwidth of BW_{max}, where each operand's size is b_{reg}. The parameters are summarized in Table 1.

Table 1. Hardware Parameters.

Parameter	Description
N_{PE}	Number of PEs (in-order processing elements)
$S_{\$}$	Cache size *[Bytes]*
N_{max}	Maximal number of thread contexts in the register file
CPI_{exe}	Average number of cycles required to execute an instruction assuming a perfect (zero-latency) memory system *[cycles]*
f	Processor frequency *[Hz]*
$t_{\$}$	Cache latency *[cycles]*
t_m	Memory latency *[cycles]*
BW_{max}	Maximal off-chip bandwidth *[GB/sec]*
b_{reg}	Operands size *[Bytes]*

Clearly, the characteristics of the workload have a great impact on the attainable performance. Recall that our focus is on data-parallel workloads which can be parallelized to numerous independent threads. For benchmarks in this general family, there are three key parameters that impact performance: (1) the *scalability* of the workload, captured by the number of threads that can execute (or be ready to execute) concurrently, n; (2) the *compute intensity* of the workload, captured by the ratio of memory instructions out of the total number of instructions, r_m; and (3) the *locality* of the workload, captured by the thread cache hit rate function, $P_{hit}(s, n)$, where s is the cache size and n is the number of threads that share the cache. Note that the latter captures the hit rate in the shared (L2/L3) cache; a high hit-rate in the L1 cache, if such exists, is manifested as a higher compute-to-memory ratio. The workload characteristics are summarized in Table 2.

Table 2. Workload Parameters.

Parameter	Description
n	Number of threads that execute or are in ready state (not blocked) concurrently
r_m	Fraction of instructions accessing memory out of the total number of instructions [$0 \le r_m \le 1$]
$P_{hit}(s, n)$	Cache hit rate for each thread, when n threads are using a cache of size s

B. Performance and Power Equations

We now use the parameters defined in Table 1 and Table 2 to analyze expected performance. In this context, we make the simplifying assumption that the workload parameters are fairly static, and do not vary much over time or space (i.e., at different threads of the same application). We therefore use their *average* values in the equations below. When validating our analysis using simulations (Section V.C), we shall see that this assumption holds for most of the benchmarks considered, with few exceptions.

Given each thread's cache hit rate function and the cache and memory latencies as defined in Section III.A, we can compute the average number of cycles needed for data access, denoted t_{avg}:

$$t_{avg} = P_{hit}(S_\$, n) \cdot t_\$ + \left(1 - P_{hit}(S_\$, n)\right) \cdot t_m \quad (1)$$

Any given thread needs to stall once every $1/r_m$ instructions on average, and wait until the data it accesses is received from memory. During this stall time, the PE is left unutilized, unless other threads are available to switch-in. The number of additional threads needed to fill the PE's stall time is $\frac{t_{avg}}{CPI_{exe} / r_m}$, and hence $N_{PE} \cdot \left(1 + t_{avg} \cdot \frac{r_m}{CPI_{exe}}\right)$ threads are needed in order to fully utilize the machine.

Given a workload with $n \le N_{max}$ threads, the *processor utilization,* $0 \le \eta \le 1$, (the average utilization of all PEs), is:

$$\eta = \min\left(1 \ , \ \frac{n}{N_{PE} \cdot \left(1 + t_{avg} \cdot \frac{r_m}{CPI_{exe}}\right)}\right) \quad (2)$$

The minimum in (2) captures the fact that after all execution units are saturated, there is no gain in adding more threads to the pool. If we ignore bandwidth limitations, the expected performance is simply $N_{PE} \cdot \frac{f}{CPI_{exe}} \cdot \eta$ OPS (Operations Per Second).

However, since bandwidth to external memories is limited, this performance level cannot always be reached. In fact, off-chip bandwidth is a principal bottleneck that often limits performance. For a given workload, a given number of threads, and given performance (in OPS), the off-chip bandwidth generated can be expressed as:

$$BW = Performance \cdot r_m \cdot b_{reg} \cdot \left(1 - P_{hit}(S_\$, n)\right) \quad (3)$$

Hence, given an off-chip bandwidth limit BW_{max}, the maximal performance achievable by the machine is $BW_{\max} / \left(r_m \cdot b_{reg} \cdot \left(1 - P_{hit}(S_\$, n)\right)\right)$. Thus, performance can be expressed using the following equation:

$$Performance\ [OPS] = \min\left[\left(N_{PE} \cdot \frac{f}{CPI_{exe}} \cdot \eta\right), \left(\frac{BW_{\max}}{r_m \cdot b_{reg} \cdot (1 - P_{hit}(S_\$, n))}\right)\right] \quad (4)$$

With power and energy consumption becoming key factors in practically all modern computer systems, performance under a given power envelope and power efficiency become primary design targets. Power consumption can be modeled as $Power_{leakage}$+$Performance \cdot EPI$, where *Performance* is given by (4), and *EPI* is the average consumption of Energy Per Instruction. Using the notations in Table 3, power consumed can be expressed using the following equation:

$$Power = Power_{leakage} + Performance \cdot \left[e_{ex} + r_m \cdot \begin{pmatrix} P_{hit}(S_\$, n) \cdot e_\$ + \\ \left(1 - P_{hit}(S_\$, n)\right) \cdot e_{mem} \end{pmatrix}\right] \quad (5)$$

Notice that as the number of concurrent threads grows, the hit rate for each of them degrades, and thus more accesses are served from memory. Since memory access is significantly more energy-costly than access to the on-die cache, the EPI increases with the number of threads. This effect is more significant for architectures which achieve their performance via a very high thread count.

Table 3. Hardware Power Parameters.

Parameter	Description
e_{ex}	Energy per operation *[j]*
$e_\$$	Energy per cache access *[j]*
e_{mem}	Energy per memory access *[j]*
$Power_{leakage}$	Leakage power *[W]*

IV. Performance and Power Curve Study

In this section we study how various workload characteristics affect the performance (Section A) and power (Section B) curves. For this study, we use an example machine consisting of 1024 PEs and a 16MB cache. The machine supports up to N_{max} = 65536 in-flight threads and runs at a frequency of 1GHz with a CPI_{exe} of 1 cycle. The machine requires 1 cycle to access its on-chip cache and 200 cycles to access off-chip memory, whose bandwidth is 200GB/sec. We assume single precision calculation (i.e., an operand size of 4 bytes).

We begin with synthetic workloads to enable a clean study of trends and the effect of different parameters on the performance plot. These will be replaced with real workloads in Section V. We use the next simple cache hit rate function, first suggested by Jacob et al. [19]:

$$P_{hit}(S_\$, n) = 1 - \left(\frac{S_\$}{n \cdot \beta} + 1\right)^{-(\alpha - 1)} \quad (6)$$

This function is based upon the well known empirical power law from the 70's (also known as the 30% rule or the √2 rule) [20]. In (6), workload locality increases when increasing α or decreasing β. The parameter β also accounts for the degree of sharing among the threads: in case much of the cache is shared, each thread can utilize a larger portion of the cache, which is represented by a small value of β.

A. *Performance Curve Study*

1) Parameters Sensitivity: Fig. 1 shows the performance vs. the number of threads, n, available in the workload, for synthetic benchmarks with different cache hit rate functions (i.e., data locality).

In Fig. 1, three performance regions are clearly evident: In the leftmost region, as long as the cache capacity can effectively serve the growing number of threads, increasing the number of threads improves performance, as more PEs are utilized. This is the *cache-efficiency zone*. At some point, the cache becomes too small for the growing stream of access requests, so memory latency is no longer masked by the cache and performance improves more moderately, or even takes a dip into a valley. As the number of available threads again increases, the *multithread efficiency zone* (on the right) is reached, where adding more threads improves performance up to the maximal performance of the machine, or up to the bandwidth wall. In Section B we show that power considerations also limit the achievable peak. Only scalable workloads with a high enough number of independent threads can benefit from this region.

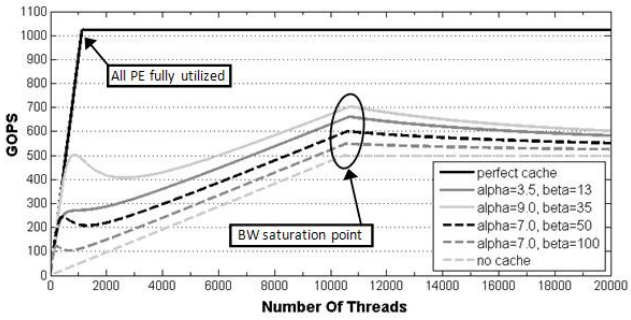

Fig. 1. Performance vs. number of threads for benchmarks with different cache hit rate functions ($P_{hit}(s,n)$), increasing from 0 (no cache) to 1 (a perfect cache), r_m=0.1. Performance increases with increased locality, especially in the cache efficiency zone.

Fig. 1 shows that workloads with higher locality better exploit the cache and hence expand the cache efficiency zone to the right and up. Workloads with poor locality cannot utilize the cache and hence gain performance only from increase in their thread level parallelism. Moreover, the shape of the performance curve depends on how fast the cache hit rate degrades as a function of the number of threads: The valley occurs whenever the degradation in cache hit rate is of the form $n^{(1+\varepsilon)}$ for some positive ε, representing a super-linear dependency of the hit rate degradation in the number of threads. (This degradation rate can be computed by deriving the performance formula (4) as a function of n). We see that when this condition is not met (e.g., in the dark-gray solid curve; α=3.5, β=13), there is no valley between the two regions.

Another point to notice in Fig. 1 is that, once the bandwidth requirements exceed the capacity, performance actually starts to degrade. This happens because P_{hit} is affected by the number of threads— the more in-flight threads there are, the less cache is available to each of them. Therefore, when the off-chip bandwidth wall is met, adding more threads only degrades performance due to increasing off-chip pressure.

Fig. 2 shows how the compute intensity of the workload affects the shape of the performance plot. When there are more computation instructions per memory access, (a smaller r_m), performance climbs more steeply with additional threads. This is because as more instructions are available for each memory access, fewer threads are needed to fill the stall time resulting from waiting for memory. Thus, compute-intense applications can reach peak performance with less parallelism and smaller bandwidth requirements. All in all, a high compute/memory ratio decreases the need both for caches and for scaling the application to many threads.

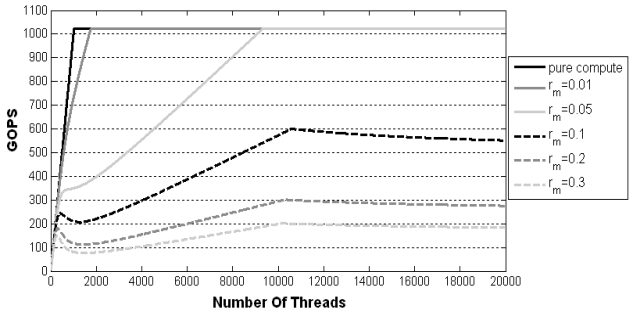

Fig. 2. Performance in a limited BW environment for benchmarks with different percentages of memory instructions (r_m), α=7, β=50. Performance increases with the compute intensity, i.e., as r_m decreases.

2) Workloads Families: Looking at the curves of Section IV.A.1, we observe that, by-and-large, workloads exhibit three different behavior patterns depending on their parameters. Fig. 3 schematically plots these three examples (stopping before the bandwidth saturation point). Simulation results (Section V) will later validate that all three classes of behavior can be found in "real" workloads.

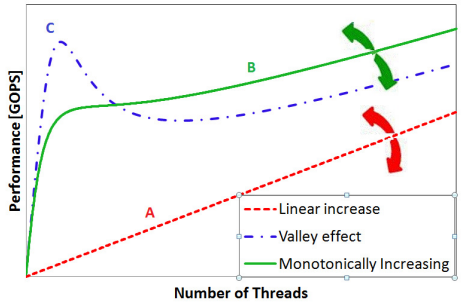

Fig. 3. Performance curve for the 3 types of workloads: (A) Workloads with a constant hit-rate in the cache exhibit linearly increasing performance (the slope depends on the hit rate), (B) workloads exhibiting a nonlinear but monotonically increasing performance (their hit-rate is mildly reduced as more and more threads share the cache), (C) workloads exhibiting a performance valley (their hit rate is sharply reduced as the cache is shared by more and more threads).

The Performance of workloads of class A (dashed line) grows linearly with the number of threads. These workloads have a constant cache hit rate which is independent of the number of threads. This may happen for example due to poor locality (light-gray dashed curve in Fig. 1, hit rate = 0%) in one extreme case, or due to full sharing of data among all threads (assuming all data is cached, black curve in Fig. 1) in the other extreme case. Workloads of this class can efficiently use either classic multithreading-based or classic cache-based architectures, depending on their (constant) cache hit-rate.

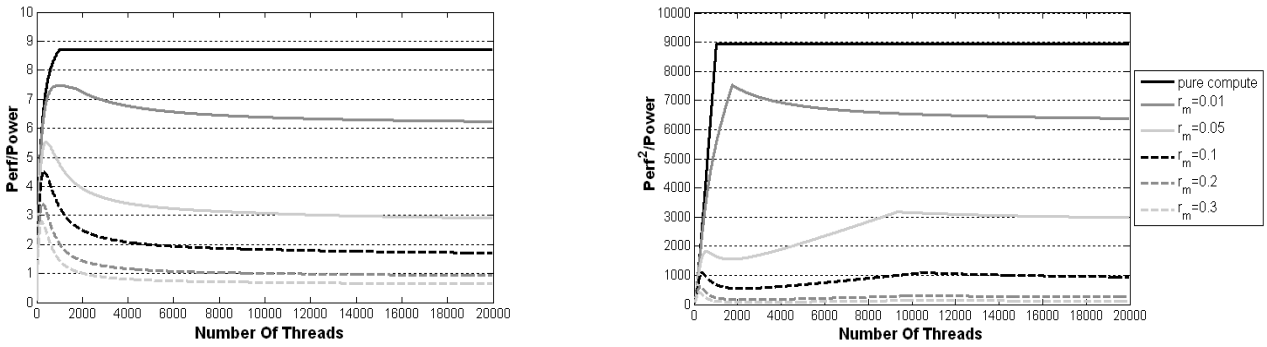

Fig. 4. Efficiency metrics: (a) *Performance/Power* (1/EPI) and (b) *Performance²/Power* (1/(energy·delay)) of the unified machine, α=7, β=50 . The former is always better in the cache efficiency (left hand) zone, whereas for the latter, the favorable archetype varies according to the workload characteristics.

The other two workload classes present non-linear behavior. Both have an operation zone where the cache is more effective and an operation zone where multithreading is more effective, but they differ in the area between these two zones. Workloads of class B (solid line) exhibit a monotonically increasing performance. They are characterized by a sub-linear degradation of their cache hit rate function in the number of threads. The transition from the cache effective zone to the multi-threading effective zone does not incur a performance lose but rather a reduced rate of performance improvement. Workload of this class will perform better with aggressive multi-threading - at least in an unrestricted environment where no memory or bandwidth constraints exist.

Workloads of class C (dotted line) present a valley-like behavior, and are characterized by a super-linear degradation of the hit rate in the number of threads. Optimizing the architecture for such workloads is especially challenging, because by trying to leverage a combination of the two approaches, these workloads might end up in a performance zone inferior to their achievable peaks either in cache-only or multithreaded-only architectures.

Lastly, note that the x axis, *n*, represents the number of threads that can actually run at a time, and does not include ones that are blocked, either on I/O or on synchronization. In workloads with extensive synchronization or I/O activity, *n* will be limited, so the plotted performance curve will be pruned somewhere along the x axis. Likewise, recall that the achievable peak in the multithread zone is limited by the maximal bandwidth (as seen in Fig. 1) and, as we show next (in Section B), by the engine's power envelope.

B. Power Efficiency and Power Envelope

In the following study of power costs, we take as an example an energy per operation (e_{ex}) of 0.1nJ, and factors of 5 and 50 for accesses to cache ($e_{\$}$) and memory ($e_{mem}$), respectively. We note that this is only one such example; using the analytical model, other ratios can be plugged in to derive results.

The performance versus power tradeoff is typically studied using one of the following two efficiency metrics: the normalized power consumption per instruction, which is captured by *Performance/Power*, or energy·delay which is captured by *Performace²/Power*. Fig. 4 presents these two metrics for the benchmarks of Fig. 2. We see that in terms of *Performance/Power* metric (Fig. 4(a)), using caches is always preferable as they enable serving most of the data accesses from the cache rather than memory. For the more performance-oriented metric (Fig. 4(b)), some of the workloads favor caching, whereas others favor multithreading. For example, for the dark-gray dashed curve (r_m=0.2), the performance boost via multithreading comes at too high a power toll, and hence caching is the preferable approach in this case. For the light-gray solid curve (r_m=0.05), the performance boost from multithreading exceeds the increase in power consumption, and hence multithreading is the favorable approach.

The results above were obtained assuming that power consumption is not constrained. In practice, however, machines have a limited *power envelope* under which they need to function. Fig. 5 assumes a power envelope of 300 Watts, and revisits the performance curve of Fig. 2 under this constraint.

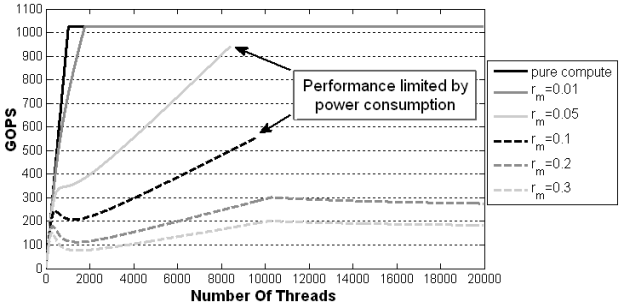

Fig. 5. Performance under a power envelope of 300 Watts, α=7, β=50. Power constraints limit performance achievable via the use of multithreading since it increases the frequency of accesses to main memory, which are more expensive in terms of power.

Recall that increasing the number of threads increases the energy spent on each instruction. Thus, the number of threads that can be run within a given power envelope is bounded, and the achievable performance in a limited power environment is smaller than the theoretical (unconstrained) peak performance of the machine.

In Section III.B we saw that for highly parallel workloads, the best performance can be achieved by a high thread count, which provided the motivation for aggressively multi-threaded machines with simple cores and no caches. Nevertheless, we now see that this approach is limited by power constraints, and that caches can help reduce the energy costs per instruction, especially for workloads with good locality properties. This is part of the reason why new GPU engines incorporate more on-chip memory than in the past.

V. PERFORMANCE SIMULATION

A. Simulator

We use an in-house simulator, MTM$im [21], specifically designed for simulating graphics-oriented architectures with numerous cores. MTM$im uses the Pin binary instrumentation tool [22]. It models a highly multi-threaded machine with a parameterized number of cores and a large shared cache (of parameterized size). Each core is a simple processing element, such that every instruction takes a predefined number of cycles to execute unless it accesses the memory hierarchy. MTM$im implements a shared last level cache and parameters define the latencies for memory access in cases of hit or miss. The simulated machine saves a large number of thread contexts, (typically significantly larger than the number of cores), such that when one thread is blocked on memory access (i.e., cache miss), another thread is swapped in and utilizes the machine. We use a simple round-robin scheduling policy among all available threads.

MTM$im decouples timing/scheduling of each instruction from its functional implementation. The simulated machine engine determines *when* every instruction of every thread is executed via its timing model, and uses Pin only to determinate *what* each instruction does. Since the timing simulator determines when each thread advances, this approach captures dynamic and transient time-dependent effects such as fine-grained data sharing, thread synchronization, and in general, inter-thread communication and inter-thread interactions. It further exposes dynamic variability within the benchmark, which is not captured by our analytical model. CMPSched$im [23] and GEMS [24] both take the same approach of decupling simulation functionality and timing.

B. Workload and System Parameters

We study workloads from the PARSEC benchmark kit [7], concentrating on those with relatively high scalability: *blackscholes*, *raytrace*, *swaptions*, *canneal*, *dedup*, and *bodytrack*. (Since we are interested in pushing the number of threads to hundreds, we leave out benchmarks from the kit that either have very limited scalability, or that cannot be spawned with hundreds of threads [7][14].) We chose the PARSEC kit because it represents emerging workloads, specifically modeling future CMP applications [15]. The PARSEC kit is more diverse than traditional parallel workload suites, which focused more narrowly on high-performance computing. We run all workloads using the *simmedium* inputs, and vary the number of instantiated threads from 1 up to around 2000 or to the point where adding threads is no longer effective. (We have limited the number of threads to 2000 due to the excessive memory requirements of the Pin tool when simulating such a large number of threads.) Note that the actual number of available threads, n, is often lower than the number of instantiated threads, because threads are sometimes blocked due to synchronization or system calls. For all workloads and all runs, we fast forward through the initialization phase of the benchmark, and sample only the parallel phase where all threads have been spawned.

Note that even scalability to hundreds of threads falls short of the futuristic values used in the synthetic model above. To capture the performance trends within the much smaller simulated range, we scale down the machine parameters, to 128 PEs and a 4MB shared cache. Cache hit time is 1 cycle and memory latency is 200 cycles. The number of in-flight thread contexts is set to 10000 so that it is not a limiting factor.

C. Results

Table 4. Memory instruction ratios (r_m) for PARSEC's workloads.

Benchmark	Percent Of Memory Instructions ($100 \cdot r_m$)
blackscholes	32.72
swaptions	43.26
raytrace	64.19
dedup	36.55
canneal	34.23
bodytrack	29.33

In order to validate our analytical study, we extract the actual average workload parameters (effective number of threads n, hit rate P_{hit}, and compute/memory ratio r_m). We then substitute these values in the equations of the analytical model (Section III.B), and compare the performance values predicted by the model to the performance values measured by the simulator. Fig. 6 shows the simulated performance results and the performance predicted by the analytical model for the different workloads. For each simulation run, we plot a data point whose x coordinate reflects the average number of available threads (running or ready) over all times in the run, and the y coordinate reflects performance in the same run. Fig. 6 also shows the average cache hit rate extracted from simulations for each applications, as a function of the number of threads. The ratio of memory instructions, r_m, for each of the benchmarks is given in Table 4. We found that for all considered workloads, r_m is practically the same for any number of threads the workload is parallelized to.

We observe that the analytical model predicts performance accurately for our target applications, namely, symmetric, parallel workloads. This essentially shows that we can represent a workload very accurately using three numbers – n, P_{hit}, and r_m, and that our analytical model, despite being simple, effectively captures the interplay of these three parameters with the architecture. Moreover, the close correspondence between the model and the simulations also shows that using average values of hit rate and compute-to-memory ratio is a reasonable approximation in most cases. For asymmetric workloads like *bodytrack*, where the workload parameters vary in space (i.e., at different threads) and in time, our analysis is less accurate. We note that a more detailed simulator might well amplify the deviation of the model prediction from the simulation results. Nevertheless, such deviations would not undermine our *qualitative* study, which does not seek to obtain *quantitative* expected performance numbers on any given hardware.

We further see that the PARSEC workloads span the range of behaviors depicted in Fig 3. Some workloads (*blackscholes*, *swaptions* - financial workloads) exhibit a valley-like shape. Indeed, as can also be seen in Fig 6, both incur superlinear degradations in their hit rates as the number of threads increases. The two workloads, however, differ in the gradient

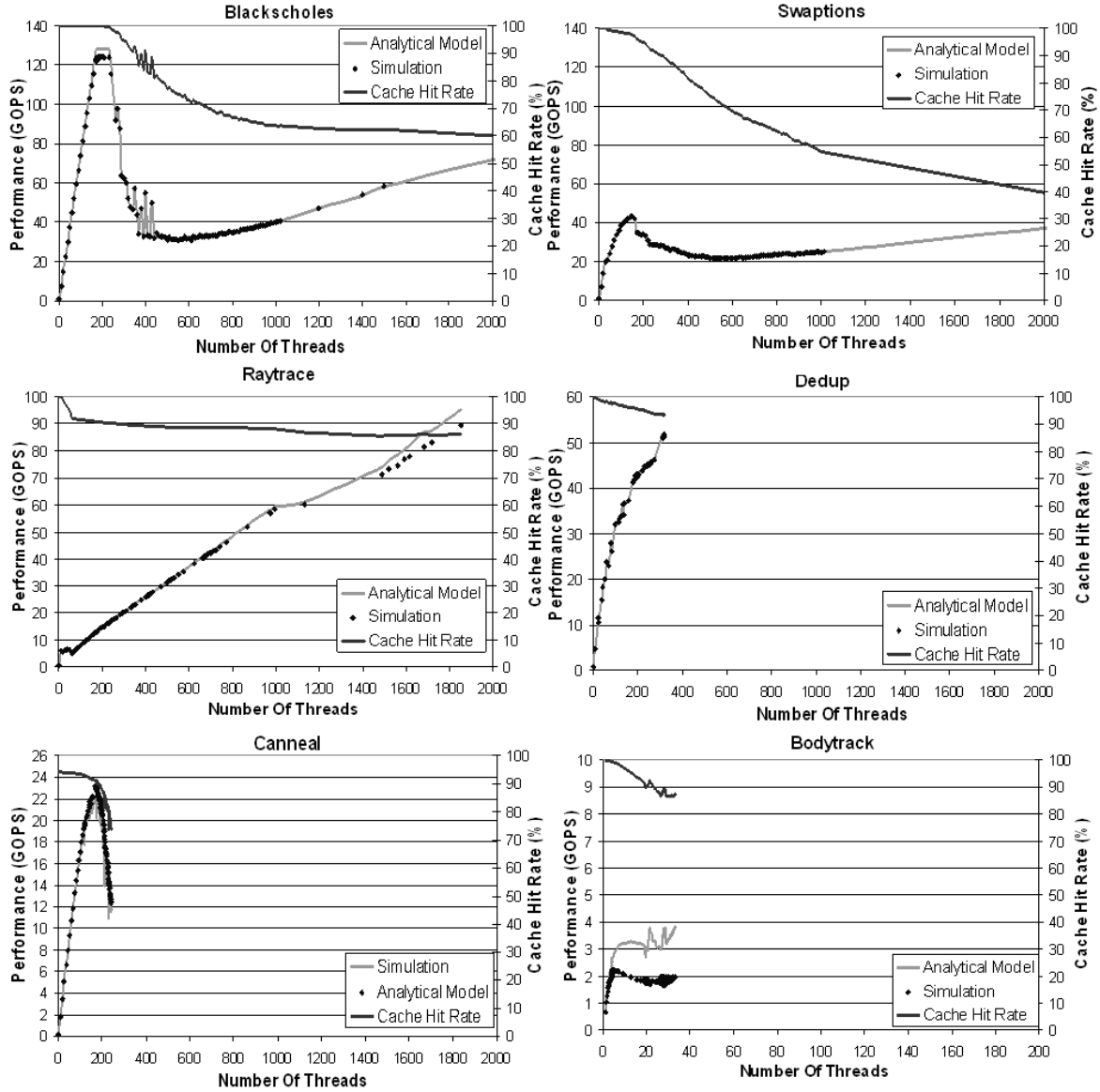

Fig. 6. Performance vs. number of available threads as extracted from simulation and as predicted by the analytical model with actual workload parameter values. The figures also show the average hit rate for each workload. We see that when instantiated with benchmark-specific parameter values, the analytical model provides a very accurate prediction of performance for most benchmarks; this implies that benchmark parameters vary very little both in space and in time during their parallel phase. One exception is *bodytrack,* where the number of active threads varies greatly in time, and hence our analysis, although still predicting the general trend, does not closely match the simulation results. We further observe that in embarrassingly parallel workloads like *blackscholes* and *swaptions*, (financial workloads) the valley shape is clearly exhibited, with a steep climb in the former and a mild climb in the latter. Other workloads, like *canneal* (simulated annealing) and *dedup* (compression), are not sufficiently scalable to climb out of the valley and benefit from the MT zone. Lastly, workloads like *raytrace* do not descend into a valley, and present better performance as more threads are spawned.

of their performance growth in the multithreading effective zone: being more compute-intensive, blackscholes climbs faster than swaptions, gaining more performance out of every additional thread. The higher compute intensity and better locality allow blackscholes to also reach a higher peak in the cache zone compared to swaptions. In fact, blackscholes fully utilizes all execution units of the simulated machine in the cache effective zone (hence the plateau in its peak).

On the other hand, *raytrace* (a graphics workload) does not exhibit a valley. Indeed, its performance continues to increase with every additional thread. This is because, as Fig. 6 also shows, its cache hit rate degrades sub-linearly, thanks to extensive data sharing among threads.

Other workloads, like *canneal* (simulated annealing) and *dedup* (compression workload) can only operate in the cache efficiency zone due to their limited scalability. Despite spawning numerous threads when running these benchmarks, we could not get more than 250-300 of their threads to run (or be ready to run) concurrently. Notice that this limited scalability is not due to the overhead of the synchronization primitives themselves, but rather due to true dependencies among threads, which block waiting for each other.

D. Performance as a function of hardware parameters

While in Section IV we study how workloads parameters affect the shape of the performance curve, we now study how

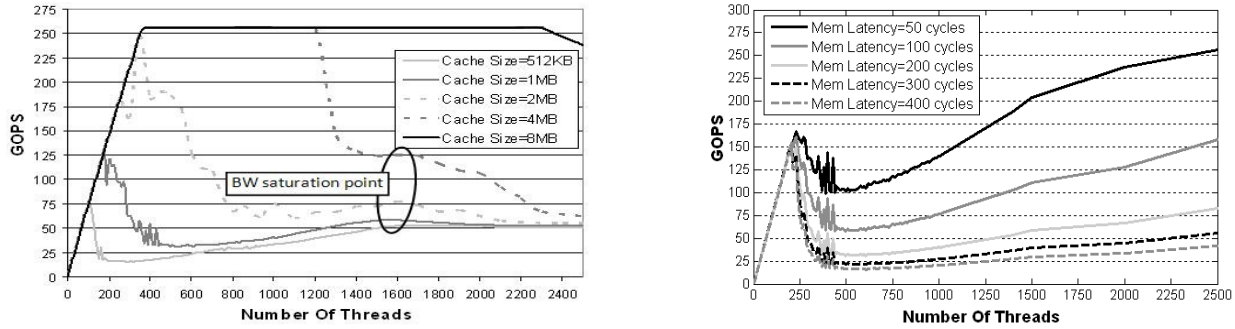

Fig. 7. Performance of the blackscholes workload across machines with (a) different cache sizes, and (b) different memory latencies.

a given workload behaves across different machines. We take blackscholes as a real example and use its real characteristics (i.e., r_m, and $P(S_\$,n)$) as extracted from simulations.

Fig. 7(a) presents the behavior of blackscholes for machines with different cache sizes. Naturally, larger caches are able to extend the cache efficiency zone up and to the right. Less intuitive is the fact that caches are also crucial in the multithreading zone. Notice that for larger caches, the peak achievable in that region is higher. This is because the achievable performance via the use of multithreading is limited by the machine bandwidth. Larger caches are able to better reduce the pressure on the external memory, therefore hindering the point of bandwidth saturation longer.

Fig. 7(b) presents the behavior of blackscholes over machines with different memory latencies. We see that longer latencies reduce the rate of the climb in the multithreading zone, since more threads are needed to mask the longer latencies to memory. Since the memory latency wall is only getting worse [2], caches will become even more critical in the future, as gaining performance out of multithreading will become increasingly hard.

VI. Conclusions

This paper sought to shed some new light on the two fundamental approaches to achieving high-performance in the multi-core era, namely caching and aggressive multithreading. To this end, we presented a simple closed-form model, validated by simulations. To the best of our knowledge, ours is the first analytical model to account for both memory-masking techniques. As such, our model captures current architectures that employ either one of the approaches, as well as novel high performance engines, like Nvidia's Fermi, which leverage both.

Our model facilitates reasoning about complex phenomena that arise when both approaches are in play. We used it in a qualitative study of representative workloads on characteristic high-performance architectures. We observed that as the number of threads scales up, different benchmarks exhibit very different performance curves. In some cases, perhaps counter-intuitively, performance is not monotonic with the number of threads.

Finally, we believe that our model can direct further research on ways to address the memory wall problem in high-performance engines.

VII. References

[1] General-Purpose Computation Using Graphics Hardware, http://www.gpgpu.org/

[2] M. V. Wilkes, "The memory gap," Keynote address, *Workshop on Solving the Memory Wall Problem*, *ISCA 2000*.

[3] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, et al., "Larrabee: a many-core x86 architecture for visual computing," *SIGGRAPH 2008*.

[4] NVIDIA GeForce series GTX280, 8800GTX, 8800GT, http://www.nvidia.com/geforce

[5] ATI Mobility RadeonTM HD4850/4870 Graphics-Overview, http://ati.amd.com/products/radeonhd4800

[6] "NVIDIA's next generation CUDA compute architecture: Fermi," Nvidia Corporation, 2009.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," *PACT 2008*.

[8] A. Agrawal, "Performance tradeoffs in multithreaded processors," *IEEE Trans. on Parallel and Distributed Systems*, 1992.

[9] R. H. Saavedra-Barrera, and D. E. Culler, "An analytical solution for a markov chain modeling multithreaded," technical report CSD-91-623, UC Berkeley, 1991.

[10] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood, "Analytic evaluation of shared-memory systems with ILP processors," *ISCA 1998*.

[11] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," *ISCA 2009*.

[12] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W. –M. Hwu, "An adaptive performance modeling tool for GPU architectures," *PPoPP 2010*.

[13] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser, "Many-Core vs. Many-Thread machines: stay away from the valley," *Computer Architecture Letters*, vol. 8, April 2009.

[14] M. Bhadauria, V. M. Weaver, and S. A. McKee, "Understanding PARSEC performance on contemporary CMPs," *IISWC-2009*.

[15] C. Bienia, S. Kumar, and K. Li, "PARSEC vs. SPLASH-2: a quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," *IISWC-2008*.

[16] "Opencl - the open standard for parallel programming of heterogeneous systems," http://www.khronos.org/opencl, 2009.

[17] G. Diamos, A. Kerr, and M. Kesavan, "Translating GPU binaries to tiered SIMD architectures with Ocelot," technical report GIT-CERCS-09-01, Georgia Institute of Technology, 2009.

[18] "CUDA Programming Guide 2.0," Nvidia Corporation, 2008.

[19] B. L. Jacob, P. M. Chen, S. R. Silverman, and T. N. Mudge, "An analytical model for designing memory hierarchies," *IEEE Transactions on Computers*, vol. 45, no 10, October 1996.

[20] C. K. Chow, "Determination of cache's capacity and its matching storage hierarchy," *IEEE Transactions on Computers*, vol. c-25, 1976

[21] "The MTM$im Simulator," unpublished.

[22] C.K. Luk, R. Cohn, R. Muth, et al., "Pin: building customized program analysis tools with dynamic instrumentation," *PLDI 2005*.

[23] J. Moses, K. Aisopos, A. Jaleel, et al., "CMPSched$im: evaluating OS/CMP interaction on shared cache management," *ISPASS 2009*.

[24] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, et al., "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Computer Architecture News*, Vol. 33, No. 4, November 2005.