Operating System Support for Mitigating Software Scalability Bottlenecks on Asymmetric Multicore Processors

Juan Carlos Saez
Complutense University,
Madrid, Spain
jcsaezal@fdi.ucm.es

Alexandra Fedorova
Simon Fraser University,
Vancouver BC, Canada
fedorova@cs.sfu.ca

Manuel Prieto
Complutense University,
Madrid, Spain
mpmatias@dacya.ucm.es

Hugo Vegas
Complutense University,
Madrid, Spain
hugovegas@fdi.ucm.es

ABSTRACT

Asymmetric multicore processors (AMP) promise higher performance per watt than their symmetric counterparts, and it is likely that future processors will integrate a few *fast* out-of-order cores, coupled with a large number of simpler, *slow* cores, all exposing the same instruction-set architecture (ISA). It is well known that one of the most effective ways to leverage the effectiveness of these systems is to use fast cores to accelerate sequential phases of parallel applications, and to use slow cores for running parallel phases. At the same time, we are not aware of any implementation of this *parallelism-aware* (PA) scheduling policy in an operating system. So the questions as to whether this policy can be delivered efficiently by the *operating system* to unmodified applications, and what the associated overheads are remain open. To answer these questions we created two different implementations of the PA policy in OpenSolaris and evaluated it on real hardware, where asymmetry was emulated via CPU frequency scaling. This paper reports our findings with regard to benefits and drawbacks of this scheduling policy.

Categories and Subject Descriptors

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

General Terms

Performance, Measurement, Algorithms

Keywords

Asymmetric multicore, Scheduling, Operating Systems

CF'10, May 17–19, 2010, Bertinoro, Italy.

1. INTRODUCTION

An asymmetric multicore processor (AMP) includes cores exposing the same instruction-set architecture, but differing in features, size, speed and power consumption [2, 8]. A typical AMP would contain a number of simple, small and low-power *slow* cores and a few complex, large and high-power *fast* cores. It is well known that AMP systems can mitigate scalability bottlenecks in parallel applications by accelerating sequential phases on fast cores [2, 7, 12].

To leverage this potential of AMP systems, threads must be mapped to cores in consideration of the amount of parallelism in the application: if an application is highly parallel its threads should be mapped to slow cores, but if the application is sequential or is executing a sequential phase its thread should be mapped to a fast core. A natural place for this *Parallelism-Aware* (PA) policy is in the operating system. This way, many applications can reap its benefits, potentially without requiring any modifications, and the sharing of scarce fast cores among multiple applications can be fairly arbitrated by the operating system. To the best of our knowledge, there are no OS-level implementations of the PA scheduling policy. As a result, many questions regarding the effectiveness and practicality of this policy remain open.

One open question is *how can the operating system effectively detect sequential phases in applications?* In some applications unused threads block during the sequential phase, and by monitoring the application's runnable thread count, which is exposed to the OS by most threading libraries, the scheduler can trivially detect a sequential phase. In other applications, however, unused threads busy-wait (or spin) during short periods of time, and so the OS cannot detect these phases simply by monitoring the runnable thread count. To address these scenarios we designed PA Runtime Extensions (PA-RTX) – an interface and library enhancements enabling the threading library to notify the scheduler when a thread spins rather than doing useful work. We implemented PA-RTX in a popular OpenMP runtime, which required only minimal modifications to support them, but the extensions are general enough to be used with other threading libraries.

Another open question is the overhead associated with the PA policy. Any policy that prioritizes fast cores to specific

threads is bound to generate migration overheads – a performance degradation that occurs when a thread is moved from one core to another. Performance degradation results from the loss of cache state accumulated on the thread's old core. Upon evaluating these overheads we found that they can be significant (up to 18%) if the fast core is placed in a different memory hierarchy domain from slow cores, but a hardware configuration where a fast core shares a memory hierarchy domain with several slow cores coupled with a topology-aware scheduler practically eliminates these overheads.

We evaluate the PA policy on a real multicore system, where "slow" cores were emulated by reducing the clock frequency on the processors, while "fast" cores were configured to run at regular speed. We find that the main benefits from the PA policy are derived for multi-application workloads and when the number of fast cores relative to slow cores is small. In this case, it delivers speedups of up to 40% relative to the OpenSolaris default asymmetry-agnostic scheduler. Previously proposed asymmetry-aware algorithms, which we used for comparison, also do well in some cases, but unlike our parallelism-aware algorithms they do not perform well across the board, because they fail to consider the parallelism of the application.

The key contribution of our work is the evaluation of the operating system technology enabling next-generation asymmetric systems. We are not aware of previous studies investigating the benefits and drawbacks of the PA scheduling policy implemented in a real OS. Our findings provide insights for design of future asymmetry-aware operating systems and asymmetric hardware alike.

The rest of the paper is structured as follows. Section 2 presents the design and implementation of the PA scheduling algorithm. Section 3 presents experimental results. Section 4 discusses related work. Section 5 summarizes our findings and discusses future work.

2. DESIGN AND IMPLEMENTATION

In Section 2.1 we describe two parallelism-aware algorithms proposed in this work: PA and MinTLP. In Section 2.2 we describe the runtime extensions to PA (PA-RTX). A brief description of other asymmetry-aware algorithms that we use for comparison is provided in Section 2.3.

2.1 PA and MinTLP algorithms

Our algorithms assume an AMP system with two core types: *fast* and *slow*. Previous studies concluded that supporting only two core types is optimal for achieving most of the potential gains on AMP [8]; so we expect this configuration to be typical of future systems. More core types may be present in future systems due to variations in the fabrication process. In that case, scheduling must be complemented with other algorithms, designed specifically to address this problem [18].

The goal of the algorithm is to decide which threads should run on fast cores and which on slow cores. In MinTLP, this decision is straightforward: the algorithm selects applications with the smallest thread-level parallelism (hence the name MinTLP) and maps threads of these applications to fast cores. Thread-level parallelism is determined by examining the number of runnable (i.e., not blocked) threads. If not enough fast cores are available to accommodate all these threads, some will be left running on slow cores. MinTLP makes no effort to fairly share fast cores among all "eligible" threads. This algorithm is very simple, but not always fair.

The other proposed algorithm, PA, is more sophisticated. It classifies threads dynamically into several categories: MP, HP, and SP. The MP (mildly parallel) category includes threads belonging to applications with a low degree of thread-level parallelism, including the single-threaded applications. The HP category includes threads belonging to highly parallel applications. The MP threads will run primarily on fast cores, and the HP threads will run primarily on slow cores. Threads of applications whose runnable thread count exceeds `hp_threshold` fall into the HP category, the remaining threads fall into the MP category.

A special class SP is reserved for threads of parallel applications that have just entered a sequential phase. These threads will get the highest priority for running on fast cores: this provides more opportunities to accelerate sequential phases. To avoid monopolizing fast cores, SP threads are downgraded by the scheduler into the MP class after spending `amp_boost_ticks` scheduling clock ticks in the SP class.

If there are not enough cores to run all SP and MP threads on fast cores, the scheduler will run some of the threads on slow cores, to preserve load balance. SP threads have a higher priority in using fast cores. The remaining fast cores will be shared among MP threads in a round-robin fashion.

The scheduler keeps track of the count of runnable threads in each applicationto detect transitions between the aforementioned classes and perform thread-to-core mapping adjustments accordingly. To avoid premature migrations and preserve load balance, PA integrates a thread swapping mechanism to perform those adjustments periodically, instead of reacting to those transitions immediately (MinTLP also integrates a similar swapping mechanism).

When the change in the thread-level parallelism cannot be determined via the monitoring of the runnable thread count, PA relies on the Runtime Extensions, described in the next Section. We must also highlight that despite the fact that our evaluation has been focused on multi-threaded single-process applications, the PA and MinTLP algorithms can be easily extended to support multi-process applications using high-level abstractions provided by the operating system, such as process sets.

Although sensitivity of the PA algorithm to its configurable parameters was studied, we are unable to provide the results due to space constraints. We found, however, that it is generally easy to choose good values for these parameters. After performing such a sensitivity study, we set `amp_boost_ticks` to one hundred timeslices (1 second) and `hp_threshold` to one greater than the number of fast cores. These values ensure acceleration of sequential phases without monopolizing fast cores.

2.2 PA Runtime Extensions

The base PA algorithm introduced so far relies on monitoring runnable thread count to detect transitions between serial and parallel phases in the application. However, conventional synchronization primitives found in most threading libraries use an adaptive two-phase approach where unused threads busy wait for a while before blocking to reduce context-switching overheads. While blocking is coordinated with the OS, making it possible to detect phase transitions, spinning is not. Reducing the spinning phase enables the OS to detect more serial phases. However, in our context

it may also lead to excessive migrations and cause substantial overheads (as soon as a fast core becomes idle PA and MinTLP will immediately migrate a thread to this core). In the event these busy-waiting phases are frequent, it is helpful to give the scheduler some hints that would help it to avoid mapping spinning threads to fast cores. To that end, we propose two optimizations, which can be implemented in the threading library (applications themselves need not be changed).

2.2.1 Spin-then-notify mode

Our first proposal is a new *spin-then-notify* waiting mode for synchronization primitives. Its primary goal is to avoid running spinning threads on fast cores and save these "power-hungry" cores for other threads. In this mode the synchronization primitive notifies the operating system via a system call after a certain *spin threshold* that the thread is busy-waiting rather than doing useful work. Upon notification, the PA scheduler marks this thread as a *candidate* for migration to slow cores. We have opted to mark threads as migration candidates instead of forcing an immediate migration since this approach avoids premature migrations and allows a seamless integration with the PA and MinTLP swapping mechanisms. The synchronization primitive also notifies the scheduler when a spinning thread finishes the busy wait. In Section 3.2 we explore the advantages of using the new spin-then-notify mode. For this purpose we have modified the OpenMP runtime system to include this new mode in the basic waiting function used by high-level primitives such as mutexes or barriers.

Another potentially useful feature of this primitive may arise in the context of scheduling algorithms that map threads on AMP systems based on their relative speedup on fast vs. slow cores (see Section 4). These algorithms typically measure performance of each thread on fast and slow cores and compute its performance ratio, which determines the relative speedup [5, 9]. If a thread performs busy-waiting it can achieve a very high performance ratio, since a spin loop uses the CPU pipeline very efficiently[1]. As a result, the proposed algorithms would map spinning threads to fast cores despite they are not doing useful work. Even though these implementation issues could be solved via additional hardware support [11], a spin-then-notify primitive could help avoid the problem without needing extra hardware.

2.2.2 Exposing the master thread

We have also investigated a simple but effective optimization allowing the application to communicate to the kernel that a particular thread must have a higher priority in running on a fast core. This optimization was inspired by the typical structure of OpenMP *do-all* applications. In these applications, there is usually a *master thread* that is in charge of the explicit serial phases at the beginning, in between parallel loops, and at the end of the application (apart from being in charge of its share of the parallel loops). Identifying this master thread to the kernel enables the scheduler to give it a higher priority on the fast core simply because this thread will likely act as the "serial" thread. This hint can speed up do-all applications even without properly detecting serial phases. Our PA Runtime Extensions enable the runtime system to identify the master thread to the scheduler via a new system call. If the pattern of the application changes and another thread gets this responsibility, the same system call can be used to update this information.

To evaluate this feature, we have modified the OpenMP runtime system to automatically identify the thread executing the *main* function as the master thread to the kernel, right after initializing the runtime environment. In the same way as the implementation of spin-notify mode, only the OpenMP library needs to be modified, not requiring any change in the applications themselves. Upon receiving this notification, the PA scheduler tries to ensure that the master thread runs on a fast core whenever it is active, but without permanently binding the thread to that core as would be done with other explicit mechanisms based on thread affinities. This way, PA still allows different threads to compete for fast cores according to its policies.

2.3 The other schedulers

We compare PA and MinTLP to three other schedulers proposed in previous work. Round-Robin (RR) equally shares fast and slow cores among all threads [5]. BusyFCs is a simple asymmetry-aware scheduler that guarantees that fast cores never go idle before slow cores [4]. Static-IPC-Driven, which we describe in detail below, assigns fast cores to those threads that experience the greatest relative speedup (in terms of instructions per second) relative to running on slow cores [5]. We implemented all these algorithms in OpenSolaris. Our baseline for comparison is the asymmetry-agnostic default scheduler in OpenSolaris, referred to hereafter as Default.

The Static-IPC-Driven scheduler is based on the design proposed by Becchi and Crowley [5]. Thread-to-core assignments in that algorithm are done based on per-thread IPC ratios (quotients of IPCs on fast and slow cores), which determine the relative benefit of running a thread on a particular core type. Threads with the highest IPC ratios are scheduled on fast cores while remaining threads are scheduled on slow cores. In the original work [5], the IPC-driven scheduler was simulated. This scheduler samples threads' IPC on cores of all types whenever a new program phase is detected. Researchers who attempted an implementation of this algorithm found that such sampling caused large overheads, because frequent cross-core thread migrations were required [16]. To avoid these overheads, we have implemented a static version of the IPC-driven algorithm, where IPC ratios of all threads are measured *a priori*. This makes IPC ratios more accurate in some cases [16] and eliminates much of the runtime performance overhead. Therefore, the results of the Static-IPC-Driven scheduler are somewhat optimistic and the speedups of PA and MinTLP relative to Static-IPC-Driven are somewhat pessimistic.

3. EXPERIMENTS

The evaluation of the PA algorithm was performed on an AMD Opteron system with four quad-core (Barcelona) CPUs. The total number of cores was 16. Each core has private 64KB instruction and data caches, and a private L2 cache of 512KB. A 2MB L3 cache is shared by the four cores on a chip. The system has a NUMA architecture. Access to a local memory bank incurs a shorter latency than access to a remote memory bank. Each core is capable of running at a range of frequencies from 1.15 GHz to 2.3 GHz. Since

[1]Best practices in implementing spinlocks dictate using algorithms where a thread spins on a local variable [1], which leads to a high instruction throughput.

Table 1: Classification of selected applications.

Categories	Benchmarks
HP-CI	EP(N), vips(P), fma3d(O), ammp(O), RNA(I), scalparc(M), wupwise (O)
HP-MI	art(O), equake(O), applu(O), swim(O)
PS-CI	BLAST(NS), swaptions(P), bodytrack(P), semphy(M), FT(N)
PS-MI	MG(N), TPC-C(NS), FFTW(NS)
ST-CI	gromacs(C), sjeng(C), gamess(C), gobmk(C), h264ref(C), hmmer(C), namd(C)
ST-MI	astar(C), omnetpp(C), soplex(C), milc(C), mcf(C), libquantum(C)

Table 2: Multi-application workloads, Set #1.

Workload name	Benchmarks
STCI-PSMI	gamess, FFTW (12,15)
STCI-PSCI	gamess, BLAST (12,15)
STCI-PSCI(2)	hmmer, BLAST (12,15)
STCI-HP	gamess, wupwise (12,15)
STCI-HP(2)	gobmk, EP (12,15)
STMI-PSMI	mcf, FFTW (12,15)
STMI-PSCI	mcf, BLAST (12,15)
STMI-HP	astar, EP (12,15)
PSMB-PSCI	FFTW (6,8), BLAST (7,8)
PSMB-HP	FFTW (6,8), wupwise_m (7,8)
PSCI-HP	BLAST (6,8), wupwise_m (7,8)
PSCI-HP(2)	semphy (6,8), EP (7,8)

each core is within its own voltage/frequency domain, we are able to vary the frequency for each core independently. We experimented with asymmetric configurations that use two core types: "fast" (a core set to run at 2.3 GHz) and "slow" (a core set to run at 1.15 GHz). We also varied the number of cores in the experimental configurations by disabling some of the cores.

We used three AMP configurations in our experiments: (1) 1FC-12SC – one fast core and 12 slow cores, the fast core is on its own chip and the other cores on that chip are disabled; (2) 4FC-12SC – four fast cores and 12 slow cores, each fast core is on a chip with three slow cores; (3) 1FC-3SC – one fast core, three slow cores, all on one chip. Not all configurations are used in all experiments.

Although thread migrations can be effectively exploited by asymmetry-aware schedulers (e.g. to map sequential parts of parallel applications on fast cores), the overhead that they may introduce can lead to performance degradation. Since we also aim to assess the impact of migrations on performance we opted to select the default asymmetry-unaware scheduler used in OpenSolaris (we refer to it as Default henceforth) as our baseline scheduler. Despite Default keeps threads on the same core for most of the execution time and thus minimizes thread migrations, its asymmetry-unawareness leads it to offer much more unstable results from run to run than the ones observed for the other schedulers. For that reason, a high number of samples were collected for this scheduler, in an attempt to capture the average behavior more accurately. Overall, we found that Default usually fails to schedule single-threaded applications and sequential phases of parallel application on fast cores, especially when the number of fast cores is much smaller than the number of slow cores, such as on the 1FC-12SC and 4FC-12SC configurations.

We evaluate the base implementation of the PA algorithm as well PA with Runtime Extensions. We compare PA to RR, BusyFCs, Static-IPC-Driven, Min-TLP and to Default. In all experiments, each application was run a minimum of three times, and we measure the average completion time. The observed variance was small in most cases (so it is not reported) and where it was large we repeated the experiments for a larger number of trials until the variance reached a low threshold. In multi-application workloads the applications are started simultaneously and when an application terminates it is restarted repeatedly until the longest application in the set completes at least three times. We report performance as the speedup over Default. The geometric mean of completion times of all executions for a benchmark under a particular asymmetry-aware scheduler is compared to that under Default, and percentage speedup is reported.

In all experiments, the total number of threads (sum of the number of threads of all applications) was set to match the number of cores in the experimental system, since this is how runtime systems typically configure the number of threads for the CPU-bound workloads that we considered [19].

Our evaluation section is divided into four parts. In Section 3.1 we introduce the applications and workloads used for evaluation. In Section 3.2 we evaluate PA runtime extensions. In Section 3.3 we evaluate multi-application workloads. Finally, in Section 3.4 we study the overhead.

3.1 Workload selection

We used applications from PARSEC [6], SPEC OMP2001, NAS [3] Parallel Benchmarks and MineBench [13] benchmark suites, as well as the TPC-C benchmark implemented over Oracle Berkeley DB [14], BLAST – a bioinformatics benchmark, FFT-W – a scientific benchmark performing the fast Fourier transform, and RNA – an RNA sequencing

Table 3: Multi-application workloads, Set #2.

Workload name	Benchmarks
2STCI-2STMI-1HP	gamess, h264ref, astar , soplex, wupwise (12)
4STCI-1HP	gromacs, gamess, namd, gobmk, EP (12)
3STCI-1STMI-1PSCI	gamess, hmmer, gobmk, soplex, semphy (12)
2STCI-1STMI-1PSMI-1HP	gamess, h264ref, soplex, FFTW (6), equake (7)
3STCI-3STMI-1HP	gromacs, sjeng, h264ref, libquantum, milc, omnetpp, EP (10)
3STCI-3STMI-1PSCI	gromacs, sjeng, h264ref, libquantum, milc, omnetpp, BLAST (10)

application. For multi-application workloads we also used sequential applications from SPEC CPU2006.

We classified applications according to their architectural properties: memory-intensive (MI) or compute-intensive (CI), as well as according to their parallelism: highly parallel (HP), partially sequential (PS) and single-threaded (ST). Memory-intensity was important for fair comparison with Static-IPC-Driven. CI applications have a higher relative speedup on fast cores [16] and so it was important to include applications of both types in the experiments. Parallelism class was determined by tracing execution via OpenSolaris' DTrace framework and measuring the fraction of time the application spent running with a single runnable thread. Parallel applications where this fraction was greater than 7% were classified as PS, whereas the rest were classified as HP. The ST class includes sequential applications. Table 1 shows the classification of our selected applications according to these classes. The text in parentheses next to the benchmark name indicates the corresponding benchmark suite: O –SPEC OMP2001, P– PARSEC, M – Minebench, N– NAS, C – SPEC CPU2006, and NS – other benchmarks not belonging to any specific suite.

By default, all OpenMP applications were compiled with the native Sun Studio compiler. In order to evaluate PA Runtime Extensions (Section 3.2) we had to modify the OpenMP runtime system but the source code for the Sun Studio OpenMP runtime system was not available to us. For that reason, we resorted to using the Linux version of the GCC 4.4 OpenMP runtime system in OpenSolaris[2]. Nevertheless, we observed that the performance of OpenMP applications with Sun Studio and GCC is similar.

Both OpenMP and POSIX threaded applications used in section 3.3 and 3.4 run with adaptive synchronization modes; as such sequential phases are exposed to the operating system in both cases. In these sections we do not use runtime extensions with parallelism-aware algorithms. All OpenMP applications run with the default adaptive synchronization mode used by GCC 4.4 unless otherwise noted (Sun Studio can be easily configured to use a similar adaptive mode). POSIX threaded applications (such as `BLAST` or `bodytrack`) use full blocking modes on all synchronization primitives but on those related to POSIX standard mutexes and synchronization barriers, where an adaptive implementation is provided by OpenSolaris. Unlike OpenMP applications, threads of POSIX applications spin for shorter periods of time before blocking on those adaptive synchronization primitives (these are the default parameters used in OpenSolaris).

[2]Using such a version of the runtime system required augmenting OpenSolaris with a Linux compatible `sys_futex` syscall)

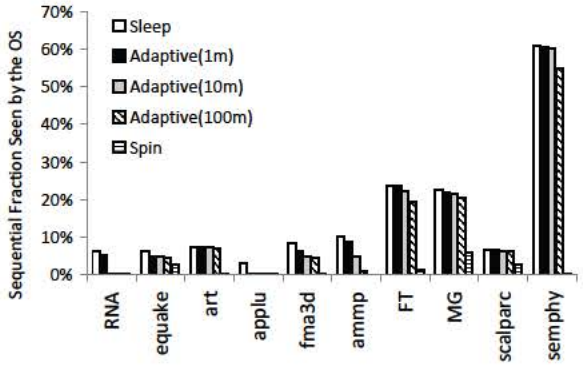

Figure 2: Variations in the sequential fraction seen by the OS when varying the synchronization mode and *blocking threshold*.

For Section 3.2 we selected ten OpenMP applications: `art`, `applu`, `fma3d`, `ammp`, `FT`, `MG`, `scalparc`, `semphy` and `RNA`. These applications were chosen to cover a wide variety of sequential portions. In the overhead section we analyze ten parallel applications accross the aforementioned classes: three HPCI (`RNA`, `wupwise` and `vips`), two HPMI (`swim` and `applu`), three PSCI (`swaptions`, `bodytrack` and `BLAST`) and two PSMI applications (`TPC-C` and `FFTW`) .

For Section 3.3, we constructed two sets of multi-application workloads. The first set, shown in Table 2, comprises twelve representative pairs of benchmarks across the previous categories mentioned above. For the sake of completeness, we experimented with additional multi-application workloads with more than two applications. Table 3 shows this second set, consisting of six workloads.

3.2 PA Runtime Extensions

We begin by investigating the effect on performance when using different synchronization waiting modes under the PA scheduler. In these experiments we demonstrate that using a low blocking threshold effectively exposes sequential phases to the scheduler, but performance can also suffer if the threshold is set too low. Then we evaluate PA-RTX and show that it offers comparable performance to purely adaptive approaches and in some cases even improves it.

In the following experiment we used the 1FC-12SC configuration and tested three different waiting modes: spin, sleep and adaptive. In spin mode unused threads busy-wait for the entire time, in sleep mode, they block immediately. We studied the effects of various synchronization modes on all asymmetry-aware schedulers, but since our results showed

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.