(big idea): starting with a multi-core design, we're going to blur the line between multi-threaded and multi-core processing.
multicore, multithreaded chips are common-place today. * consider this example of one, with eight threads running on four cores.

* consider the case when one thread stalls for an off-chip memory access: it will be unable to make forward progress for dozens, possibly hundreds of cycles. * with multithreaded cores, additional co-scheduled threads are able to keep resource utilization up.

* however, as additional threads stall for off-chip accesses, one core can quickly find its thread-level parallelism exhausted, leaving its execution units idle.

* in this system, there is still plenty of parallelism; you might consider using a thread from another core, bringing it over to take advantage of the under-utilized resources. * however, the cost of migrating between cores is prohibitive: by the time a thread has moved, the memory stalls will likely be complete, and the opportunity has passed.

* we're going to break down the barriers between cores, to enable this additional sharing; our goal is very rapid, multithreading-type context switching across cores, where migration costs dozens instead of thousands of cycles. that's the big idea here, today.
### A New Design

- A CMP of SMT cores
  - SMT: Simultaneous Multithreading (e.g. POWER5)

- Hardware support for additional threads
  - On-chip, off-core thread storage of inactive threads
  - Scheduled below O.S. level

- Mechanism for rapid thread switching
  - Move thread state between execution contexts and inactive thread storage

For our new design, we'll start off with a multi-threaded, multi-core processor, like you can find today.

* Then, we'll add hardware support for additional threads: on-chip storage for the execution state of threads, beyond those actively executing within the cores. These threads are inactive, but available for later scheduling and execution.

* Rounding out the design, we'll add a mechanism for rapidly moving thread state between the new off-core thread storage, and the existing execution contexts.
Here's an outline of the rest of my talk.

Next up, I'll introduce more details of our Shared-Thread Multiprocessor.

Following that, I'll cover our evaluation methodology.

We'll see results that demonstrate two distinct benefits of low-cost thread migration, and then we'll conclude.

**Moving on, let's look at the Shared-Thread Multiprocessor in more detail.**
We've taken a CMP, where each core is two-way multithreaded, and augmented it in two ways:

* First, we've added off-core thread storage hardware, attached to the on-chip interconnect. This holds the register state of inactive threads.

* We've also added a mechanism for rapidly moving thread state between active execution contexts, and the off-core thread storage. This mechanism is integrated into the pipeline, allowing efficient movement even as other threads execute.

With this basic design in mind, we can ask ourselves: what sort of benefits do these changes enable?
What does rapid migration enable?

- Load balancing
- Quick reaction to phase changes
- Rescheduling much faster than OS
  - Some changes come faster than OS can react

The ability to rapidly migrate threads yields several distinct benefits:

First off, it enables load-balancing at very fine time scales. *Consider this example, where two dual-thread cores are executing a total of three threads. * When the thread running solo experiences a stall for main-memory access, that core is left essentially idle, while the other core is still fully loaded. With fast enough migration, we can take advantage of the load imbalance, giving each of the two non-stalled threads their own cores, temporarily.

* Second, rapid migration enables quick reaction to phase changes. Consider a multi-core execution where one core is executing an integer-bound and a floating-point-bound thread, and the other core is executing a pair of integer-bound threads. * Then, the integer-bound thread on the first core transitions to a floating-point computation phase. This leaves the cores imbalanced; with fast migration, we could re-distribute the workload to better match the functional unit demands of the new phase.

Finally, rapid migration allows for OS-style rescheduling, but on much finer time-scales than software is capable of. * This is valuable since some application-level behavior changes faster than an OS can react, due to the high cost of software context switches.
Going back to our hardware design for a moment, let's take another look at the components we've added.

The inactive thread store holds all architected state for additional threads; for each thread, this is its registers, program counter, and any additional address-space ID or control registers as dictated by the underlying system. This is a small, fixed-sized structure which is accessed non-speculatively and non-associatively; it can be implemented cheaply as a block of SRAM.

* Next, the shared-thread control unit sits at the heart of the system, collecting periodic performance samples from the various cores, and sending messages to trigger thread migration. It executes simple scheduling algorithms using minimal amounts of memory; it can be implemented using a small in-order processor core, or perhaps even a state machine.

* Finally, to each core we add thread-transfer support. This is the only performance-critical part of our additions, since it interfaces directly with the execution pipeline. To avoid degrading performance, we utilize virtual "spill" and "fill" instructions, which are injected directly into the pipeline at the rename stage; from there on, they re-use existing mechanisms to recover register values. Each spill or fill instruction reads or writes one register value from the spill/fill buffer, which is used to aggregate groups of registers into cache-block-sized units, for ease of communication over the existing interconnect.
Alright, having introduced the Shared-Thread Multiprocessor, next we'll briefly discuss our evaluation methodology.
Methodology

- Execution-driven, multi-core out-of-order processor & memory simulation (SMTSIM)
- Four-core CMP of two-way SMT cores
- Private, coherent L1 data caches
- Shared on-chip L2 cache
  - Not a prerequisite
- Workloads composed of competing subsets of SPEC2000 benchmark suite
- Hardware, benchmark details in paper

We evaluate our design using an execution-driven, out-of-order processor and memory simulator that we've constructed. We simulate a four core processor, with two-way multithreaded cores. Our cores feature private, coherent level-one caches, and a shared on-chip L2 cache. The shared L2 is not an essential feature of our design, by the way; the idea carries over to private L2 caches as well.

We compose our workloads using competing subsets of the spec2k benchmark suite; individual benchmarks run as competing threads, without any synchronization between them.

Further hardware and benchmark details are described in the paper.
Next, we'll consider our evaluation metric for a moment. We're faced with the task of evaluating changes in overall system performance, on a workload of independent threads with diverse behavior.

Overall IPC doesn't suffice, since it unrealistically biases schedules against low-IPC threads.

* We use Weighted Speedup, a well-known metric for evaluating multithreaded execution. Rather than consider a ratio of IPC sums, weighted speedup considers a sum of IPC ratios: each thread's experimental IPC is derated by its single-thread IPC, running the same workload on the same hardware configuration. While each thread's weighted speedup contribution is typically less than 1.0, all threads execute in parallel, so the values stack.

As with regular speedup, higher weighted-speedup values reflect better overall performance.
Now, we're ready to move on to our results.
We'll first consider load-balancing. We've observed that even with SMT cores, the cores are notably under-utilized. We simulated an eight-thread CMP *, on several eight-thread workloads, * and counted the cycles in which any core was unable to commit due to stalled memory instructions.

* System-wide, we found that on average, 1/2 to 1 core's worth of commit cycles were spent idle, waiting for memory instructions. With 4 cores worth of execution resources, and 8 threads worth of work, we still observed an entire virtual core's worth of idle time; it's out there, and with fast enough thread swapping, we might be able to put that idle time to better use.
Load-balancing: Policies

<table>
<thead>
<tr>
<th>Conflict</th>
<th>Runner</th>
<th>Stall-chase</th>
</tr>
</thead>
<tbody>
<tr>
<td>conflict rate</td>
<td>1.7</td>
<td>2.3</td>
</tr>
<tr>
<td>1.8</td>
<td>Low IPC</td>
<td>0.6</td>
</tr>
<tr>
<td>T T</td>
<td>T R</td>
<td>T T</td>
</tr>
<tr>
<td>T</td>
<td>X</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>R</td>
<td>T</td>
</tr>
</tbody>
</table>

- Static comparison points
  - mean-static: mean over all possible schedules
  - best-static: oracle, best possible static schedule

- We'll evaluate five-thread cases

Here, we'll describe the shared-thread policies we used to try and take advantage of the under-utilization from memory stalls.

* First, the conflict policy: during execution, each core monitors the amount of execution resource contention: cycles when instructions are unable to make forward progress due to a lack of ALUs, renaming registers, etc. When one thread stalls for a main-memory access, we consider whether its current core is substantially more conflicted than another. If a threshold is exceeded, we take the opportunity to move that thread on to greener pastures: it's not making progress anyway, due to the memory stall.

* Next, the runner policy. With runner, one thread is designated as a "runner" -- the "R" -- which moves around, and the others are pinned in place. When another core experiences a stall for off-chip memory access, the runner is conditionally migrated to fill in the idle space, and execution continues. If another core stalls, the runner is again considered for migration. The runner is "attracted" by stall activity, in a sense.

* Finally, we have the stall-chase policy. When one thread encounters a memory stall, the system considers other threads for movement. In this example, we consider low-ipc threads, though other metrics are possible. If a thread qualifies, it is migrated to the site of the stall, and execution continues. If another thread stalls, we again consider threads for migration, moving them to the site of the stall. stall-chase is similar to runner; they differ in thread selection policies.

* For comparison, we also consider static schedules, where threads never migrate. mean-static represents the mean over all possible schedules, and best-static is the best possible static schedule, using oracle knowledge.

* We'll focus on five-thread cases, since they offer the most opportunity for migration.

The key to all of these policies is reacting, very rapidly, to memory-latency-induced imbalance. So, how did they do?
**Load-balancing: Results**

- *conflict* scheduler approaches oracle *best-static*
  - Gains limited by post-migration cache effects

Here, we have the weighted speedups corresponding to these policies. We'll focus on three data sets: *mean-static* is a useful baseline, since it corresponds to choosing a schedule at random. *conflict* is the conflict-avoidance policy that we just introduced. *best-static* is a good performance target, as it represents the most harmonious workload mix available.

* Along the X-axis, each group of bars represents the results from one five-threaded workload. * The Y-axis shows overall weighted speedup for each experiment; higher bars indicate better performance.

* focus on the orange "conflict" bars, we find that the *conflict* scheduler did well overall, beating *mean-static* in every case. * on further analysis, we found that performance was limited not by the migration operation itself, but by the cache side-effects caused by the migration: just after migration, the moved threads tend to be starved for memory working-set. They spend the "golden time" when the target core is most under-utilized, themselves stalled for on-chip cache accesses.

**Moving on, next we'll consider migration benefits at a different time-scale...**
This slide is from before, * when we outlined some potential benefits from rapid thread migration. The previous results spoke to the first two points; next, we'll focus on the third.

We're going to move up from ultra-fast time scales, reacting to individual cache misses, and consider scheduler activity on fast time scales, tens of thousands to hundreds of thousands of cycles.
Rescheduling: Motivation

- STMP offers new scheduling opportunities
  - Detailed per-thread performance data available
  - Context-switches need no software action
  - Switching cost low enough to consider more often

- Rapid rescheduling
  - Find a good schedule quickly as workloads change
  - React quickly to software phase changes

- *Complements* OS-level scheduling

In contrast to a pure software thread scheduler, the shared-thread scheduler offers new opportunities: it has detailed performance data available, without the need for explicit collection software; context switches take place without any software intervention; and most importantly, the cost of contexts switches is now low enough to consider them much more often.

* we'll utilize the fast context switches to rapidly reschedule the entire multithreaded workload -- that is, continually re-evaluate which threads run on which cores. This allows us to discover good schedules quickly as the offered workloads change, such when threads are added or removed, and also allows us to react quickly to phase changes within the applications themselves.

* it's important to note that this *complements* effective OS-level scheduling, it doesn't replace it. In such a system, instead of selecting 8T to run on specific cores, the OS might assign 10 threads for hardware execution at an OS time-slice interval, and then allow the hardware to negotiate the finer cycle-to-cycle details of scheduling them across the eight execution contexts.
To evaluate rapid rescheduling, we considered a hardware implementation of the "symbiosis" policy, which has been shown by previous work to be effective at discovering good multi-threaded schedules.

"Symbiosis" operates in two phases: first, it runs a set of random schedules for short trial periods, and then it uses the best of those schedules for a much longer "run" phase, before repeating the sampling.

We implement symbiosis in our new hardware, then then evaluate it against that same policy, but restricted to sampling rates which are feasible in software; this represents a good O.S. scheduler.
Here, we can see the performance of symbiosis, over different time scales.

* The X-axis is log-scale, and shows the sampling interval used; as you move to the left, the frequency doubles with each point.

* The Y-axis shows weighted speedup; higher results are better. This is from a ten-threaded workload, so there is plenty of diversity in the schedule space.

* The horizontal blue line shows the baseline "software" performance level, while the purple trace shows the result of running the same algorithm in hardware, at successively higher frequencies.

* What we see is that, as scheduling operations increase in frequency, performance climbs noticeably, over three successive doublings of frequency. As we keep doubling the scheduling frequency, we eventually see our gains overtaken by the overhead of thread migration. The best result was at eight times the "software" capable frequency.
Conclusion

- Shared-Thread Multiprocessor
  - Extends multithreading across CMP cores
  - Low additional complexity
  - Maintains single-thread performance
- Load-balancing around off-chip stalls
  - Approached oracle static performance
- Rescheduling of highly concurrent workloads
  - Best rescheduling was at ~8x software rate
- Thank you!

Wrapping up,

We've introduced the Shared-Thread Multiprocessor, an architecture which blurs the line between multi-core and multi-threaded processing, by enabling multi-threading-style hardware context switching, across CMP cores.

* We've seen this rapid context-switch ability used to balance utilization by reacting very quickly to individual off-chip memory stalls; starting from random initial schedules, the system was able to nearly match oracle static scheduling.

* We've also used rapid context-switching to schedule highly concurrent workloads; we saw that we were able to improve the performance of software symbiosis, by running it at 8 times the frequency, in hardware.

* Thank you all for your time + welcome any questions
Previous work in this area has touched upon this potential. Balanced Multithreading, BMT, introduced virtual contexts on a single multithreaded core.

These virtual contexts hold idle threads. In BMT, threads execute normally, until one of them is found to be blocking commit due to a memory load. Then, threads are rotated among the idle and active contexts. BMT realizes benefit from over-subscribing a single core, to cover memory stalls.

The shared-thread multiprocessor extends this idea to CMPs. However, the STMP enables more than just a "flock of BMTs"; the idle thread storage is shared among cores, enabling rapid migrations between cores as well.
Goals: Best of Both Worlds

- Throughput scales well with added TLP
- Good single-thread performance (latency)
- Respond quickly to changes in workload
  - Effective parallelism not constant
- Modest hardware cost
Task: evaluate *recent* performance
- WSU: standalone IPC unavailable on-line

Interval Weighted Speedup
- Replace standalone IPC with "basis" IPC
- Use each thread's IPC, averaged over entire previous round
- Tradeoff: short-term stability vs. adaptation
- Only for scheduling decisions, not overall results

Still use WSU to report overall performance