DynaMOS: Dynamic Schedule Migration for Heterogeneous Cores

Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke
Advanced Computer Architecture Laboratory
University of Michigan, Ann Arbor, MI
{shrupad, lukefahr, reetudas, mahlke}@umich.edu

ABSTRACT
InOrder (InO) cores achieve limited performance because their inability to dynamically reorder instructions prevents them from exploiting Instruction-Level-Parallelism. Conversely, Out-of-Order (OoO) cores achieve high performance by aggressively speculating past stalled instructions and creating highly optimized issue schedules. It has been observed that these issue schedules tend to repeat for sequences of instructions with predictable control and data-flow. An equally provisioned InO core can potentially achieve OoO’s performance at a fraction of the energy cost if provided with an OoO schedule. In the context of a fine-grained heterogeneous multicore system composed of a big (OoO) core and a little (InO) core, we could offload recurring issue schedules from the big to the little core, to achieve energy-efficiency while maintaining performance.

To this end, we introduce the DynaMOS architecture. Recurring issue schedules may contain instructions that speculate across branches, utilize renamed registers to eliminate false dependencies, and reorder memory operations. DynaMOS provisions little with an OoO mode to replay a speculative schedule while ensuring program correctness. Any divergence from the recorded instruction sequence causes execution to restart in program order from a previously checkpointed state. On a system capable of switching between big and little cores rapidly with low overheads, DynaMOS schedules 38% of execution on the little on average, increasing utilization of the energy-efficient core by 2.9X over prior work. This amounts to energy savings of 32% over execution on only big core, with an allowable 5% performance loss.

Categories and Subject Descriptors
C.1.3 [Architectures]: Heterogeneous systems

Keywords
heterogeneous processors, fine-grained phase prediction,
energy-efficiency

1. INTRODUCTION
Out-of-Order (OoO) cores are ubiquitous today in mobile phones to servers alike because they can achieve high single-thread performance on general purpose code. Their ability to dynamically resolve dependencies and speculatively issue instructions out of order enables them to maximize both Instruction Level Parallelism (ILP) and Memory Level Parallelism (MLP). Unfortunately, high performance comes at the cost of increased power consumption. Conversely, In-Order (InO) cores have lower complexity, allowing them to be significantly more energy efficient (3.5X) than OoO, but at a slowdown of more than 2X [1].

To address this disparity, researchers have designed heterogeneous multi-core processors [2] in which an application is mapped to the most efficient core that meets its performance requirements. ARM’s big.LITTLE [1] combines a high-performance big (OoO) core and a low-performance but energy-efficient little (InO) core. This allows an application to achieve high single-thread performance on the OoO core for phases that can utilize it, and to switch to the energy-efficient InO core for low performance phases [3], thereby reducing the overall energy consumption. Prior work has proposed heterogeneous architectures [4, 5, 6, 7] that enable fast switching between cores at the granularity of 100s of instructions. This effectively increases the opportunities of using the more energy-efficient little.

The majority of the performance advantage of an OoO core over an InO core comes from its ability to speculate past stalled loads and other long latency instructions, eliminate false dependencies between instructions, and execute instructions out-of-order. An issue schedule, or schedule, is used to refer to the sequence in which an OoO issues instructions for execution, after resolving dependencies and resource constraints. If a sequence of instructions, or trace, tends to repeat regularly in an application in the same program context, for example loops, it follows that the schedule created by an OoO core also tends to repeat. Dynamically recreating identical schedules for the same trace results in significant wastage of energy. We found that only 19% of the OoO’s performance advantage is due to its ability to react to unexpected long latency events, by creating different schedules for the same trace. Ideally,
81% of its performance could be achieved on an similarly provisioned InO core, provided it used a best-case OoO schedule for each trace and perfectly predictable control-flow. These results corroborate with recent work [8] that credit the OoO’s ability to create good static schedules as the main reason for the performance advantages of OoO over InO.

In this paper, we propose Dynamic Migration of Out-of-order Schedule (DynaMOS) for a fine-grain, tightly coupled heterogeneous processor with a big (OoO) core and an equally provisioned little (InO) core. Our key idea is to record, or memoize, a schedule by executing an instruction trace on a big core and replay the memoized schedule on the InO core for future iterations of the trace. Since big has optimized these schedules for its architecture, executing them on an equally provisioned little will achieve similar high performance. By recording and replaying trace-schedule pairs, we can offload more execution from the big core to the energy-efficient little core, without compromising performance. Similarly, prior works reuse OoO schedules to reveal energy and/or performance benefits either on the same pipeline [9, 10] or on different, specialized pipelines [11, 12].

We overcome several design challenges en route to enabling DynaMOS. First, false register dependencies like Write-After-Write (WAW) and Write-After-Read (RAW) are handled by register renaming in OoO cores. In order to preserve program correctness, this renaming must be honored by little. Second, instructions may have been moved across branches in the memoized schedules. If a conditional branch within a trace changes its direction from what was assumed while recording the schedule, the trace must be aborted and re-executed. Third, loads that have been speculative moved above other stores that write to the same location need to be detected and handled. Finally, interrupts must be handled precisely.

To address these problems, we architect two modes of execution for the little core: an InO mode where it executes instructions in program order, and an OinO mode where it executes reordered instructions as per big’s schedule. OinO is capable of reading a cached schedule, detecting and resolving false dependencies, and speculatively issuing memory operations. This allows the OinO mode to achieve nearly the same performance as an OoO core, at a fraction of the energy cost.

This paper makes the following contributions:

- We observe that repeatability of reordered schedules allows us to offload a significant fraction of big’s execution to the more energy-efficient little at a fine-granularity. We outline DynaMOS, an architecture that can detect and memoize traces that have reusable schedules on the big such that they are readily accessible for execution on little.
- We supplement little with features that empower it to run an OoO schedule, while ensuring correct program behavior and precise exceptions handling.

We adopt ARM’s terminology of big and little for our work, although one InO little has the same issue width and functional units as big.

1OinO stands for an InO core appearing to be OoO.

2Memosization defines the main reason for the performance advantages of OoO over InO.

This enables little to nearly achieve big’s performance for memoized traces, while maintaining its energy-efficiency.

- DynaMOS achieves 38% utilization of little, with small overheads including an 8% increase in energy of little. Overall, DynaMOS contributes energy savings of 32% as compared to having only a big core, a 2.2x increase over state-of-the-art [5].

2. MOTIVATION

A superscalar big’s ability to speculatively reorder instructions comes at a cost of bulky structures like reorder buffers, reservation stations and complex issue logic. When there is high variance and uncertainty in behavior of a trace in terms of its data and control-flow, this capability of big helps in creating new optimized schedules in the event of mispredictions. However for the common case, this work is redundant, as reordered schedules are repetitive.

2.1 Memoizable Schedules

Applications exhibit phased behavior, and by matching these phases to the core that best executes them, energy-efficiency can be improved with a minimal impact on performance.

Figure 1(a) illustrates fine-grained phase behavior in a subset of h264ref’s execution, a compute-intensive benchmark from SPECInt 2006 [13] exhibiting predictable control and data-flow. Each point represents a trace in program order, with 50 instructions on average, that has been partitioned into “Run on big” and “Run on little” categories by an oracle. The oracle observes performances of the traces on big, little and classifies those that show high performance on OoO to run on big and those that show no significant performance advantage on OoO to run on little. An oracle constrained to maintain performance at 95% of big can execute at-most 11% of the application on the more energy-efficient little.

In Figure 1(a), the light-colored (x) traces exhibit a memoizable schedule running on big. In Figure 1(b) DynaMOS migrates all the memoized schedules to the OinO, enabling h264ref to run a majority (77%) of its execution order.
execution on little, while maintaining the same performance level. Increased utilization of little achieves proportional energy savings.

Figure 2 shows similar opportunities to increase execution on little across the SPEC2006 suite. With a tightly coupled big-little architecture, an oracle that assumes zero switching overhead between the two cores can schedule 25% of execution on the energy-efficient little on average, while maintaining a 5% performance loss target. Supplemented little with an OnO mode for memoized schedules increases this best-case coverage by 2.2x to 80% on average.

### 2.2 Repeatability of OoO Schedules

![Figure 3: Histogram categorizing program execution based on how memoizable its OoO schedules are (its memoizability quotient). A majority of the dynamic execution falls under the highly memoizable category.](image)

A trace’s schedule on big depends largely on true dependencies between instructions, instruction latencies, and available resources in the pipeline. Unpredictable and unexpected memory and branch behavior within a trace forces the big to dynamically recreate a different optimized schedule for different instances of the trace.

Figure 3 shows a histogram of dynamic execution seen across all benchmarks categorized by how repeatable their OoO schedules are, or their memoizability quotient. We define a trace’s memoizable quotient as the percentage of its dynamic execution where it exhibited identical schedule. For instance, the category 90-99 on X-axis encapsulates all traces that bear identical schedules for greater than 90% of their run-time. Less than 30% of the execution displays schedules with low memoizability while majority (>70%) can be categorized as having a high memoizable quotient. This tallies with our intuition, considering that majority of a program is spent executing loops and their regular data and control flow causes them to be scheduled in the same way every iteration. Moreover, the figure illustrates that most traces fall in either extremes of the memoizability spectrum, showing potential for accurate, low-cost and dynamic classification schemes.

### 3. DYNAMOS DESIGN

The aim of DynaMOS is to maximize energy efficiency of general purpose applications with an allowable 5% performance loss as compared to execution on big. Our core architecture, based on a Composite Core [4], executes a single application thread, and includes both a big OoO and little InO core with the same superscalar width and number of functional units (FU). The idea is to empower the InO core to execute the OoO core’s schedule, allowing it to achieve nearly OoO performance at a fraction of the energy consumption.

A block diagram of DynaMOS’s components is shown in Figure 4 and is further described as follows:

- **big** detects traces with repeatable schedules and stores them in a Schedule Trace-Cache (STC) [14] (Section 3.3).
- An online controller [5] (Section 3.4) observes a sequence of traces in the current context and migrates execution on the little if it predicts that the program is entering a micro-phase where either 1) a significant number of the micro-phase’s traces are memoizable or 2) the micro-phase shows minimal performance advantage on the big.
- little itself can run in one of two modes: InO and OnO. InO mode fetches instructions from the shared instruction cache and executes them in program order. In OnO mode, the core fetches and executes reordered instructions from the STC and commits them atomically (Section 3.2). In case of mispredictions within such a trace, the OnO mode reverts to InO mode and the trace is re-executed from its start in program order.

The big core plays three important roles in DynaMOS. First, **big** determines which traces are memoizable and encodes these traces in the STC. Second, it executes traces that exhibit unpredictable schedules. Lastly, it executes traces that cannot achieve OoO performance in OnO because of architectural constraints. On the other hand, traces that can be run in-order without compromising performance are executed on little in InO mode. Such traces inherently suffer low performance because of low ILP, high branch mispredicts and/or dependent cache misses.

#### 3.1 Terminology

Before proceeding, we first formally define and expand terminology used consistently in this paper.

**Trace**: A trace is a sequence of instructions between two backward branches with a fixed control flow. The use of backward branches captures circular paths such
as loops, which are repetitive and have similar program behavior, as well as functions, which require either the call or the return to be a backward branch. A unique trace is identified by a hash of its header PC (the target of the previous backward branch) and the outcomes for all interleaving conditional forward branches (TraceID). This definition is synonymous with previous works [14, 15] and represents a sequence of instructions or basic blocks that have a high likelihood of appearing together.

**Schedule:** The big dynamically reorder instructions at the issue stage by selecting the next ready instruction to execute on available resources. A schedule encapsulates the sequence of instructions for a trace in the order they were issued.

**Atomicity:** As memoized schedules disobey program order, any trace executed in OinO mode must be treated as an atomic block. Any path divergence in the trace due to a branch misbehavior, exception, or interrupt causes execution of the trace to restart in InO mode.

### 3.2 OinO Mode

DynaMOS extends little with a special mode, OinO, enabling it to achieve high performance, without the high power consumption of big. OinO mode fetches memoized trace schedules from the STC, allowing it to execute reordered instructions in-order. As an in-order pipeline, little does not track the original program order. Therefore, when operating in OinO mode, DynaMOS faces the following difficulties in executing and committing out-of-order schedules:

1) False dependencies, i.e., Write-After-Write and Write-After-Read, are handled by Register Renaming in OoO cores. The OinO mode must honor renamed registers in the schedule to maintain correctness (Section 3.2.1).

2) As traces span multiple basic blocks, its instructions may have been reordered across branches. If a branch within a trace diverges from its recorded direction, OinO may have already speculatively executed some incorrect instructions. Therefore, the pipeline must be rolled back to the correct state in case such mispeculations occur (Section 3.2.1).

3) Loads may have been speculatively moved before older stores that write to the same location, leading to aliasing. These aliases need to be detected and handled (Section 3.2.3).

4) Precise interrupts should be handled (Section 3.2.4).

Note that variance in a schedule’s data-flow in OinO mode does not constitute a violation of program order. For example, instructions might have been scheduled around a load that is typically an L1-hit, incurring a 2 cycle latency. If this load misses in the L1 for one instance of the schedule, the OinO pipeline will stall on use until the load is satisfied. This is identical to stalling for a true dependency in InO pipeline. However, OinO mode lacks the big’s ability to rebuild a dynamic schedule to reorder around the miss.

#### 3.2.1 Register Renaming

In an OoO core, every destination architectural register (AR) is renamed to a new physical register (PR) to eliminate false dependencies. big assigns a free PR from a list and remembers the AR->PR mapping in a register allocation table. Maintaining a free list of PRs is a time and energy consuming operation. Instead, OinO implements a cheaper renaming scheme as follows.

The OinO mode provides a fixed pool of PRs to each AR for renaming purposes (Figure 5b). While encoding a trace schedule, big assigns every AR a suffix, which is an index denoting its unique version number within the trace. All live-in registers are required to have a 0 suffix. Every time the register is written to, the suffix is incremented, while there are free registers in its pool. All further uses of this register are updated to reflect the most recent suffix. We refer to this as Level-1 renaming, and it is done by big.

Figure 5a illustrates an example of this mechanism. The instructions shown in “Original Assembly” depict instructions in program order. In “After Level-1 Renaming”, each register is updated to the form Ri.j, where i is the original AR index and j is the suffix. In the example, R2.0 indicates a live-in value to the trace. R2.1 (Inst #1) and R2.2 (Inst #4) denote writes to R2, yielding a total of three versions of AR 2 in this trace. The 3-wide big issues this trace for execution over four cycles as shown under “Issue Order”. Level-1 trace encoding ensures correct data-flow across instructions within a trace. However, for the subsequent trace to read its correct live-in values, the current trace’s live-outs need to be available in the corresponding suffix 0 registers.

To maintain correct data-flow across traces seamlessly, the OinO performs Level-2 renaming, using a
modified version of the rotational remap scheme used by DIF [11]. Each AR is mapped to a fixed size circular buffer of PRs, shown as the Physical Register File (PRF) in Figure 5b. The head and tail pointers to this buffer are pointed to by the Global Last-Written (GLW) and Local-LW (LLW) registers respectively. On a register read, little reads from the PR pointed to by the modulo addition of the AR’s suffix and GLW. The GLW thus represents the index of the 0-suffix PR (live-in value to the trace) and the LLW points to the latest index written by the current trace. Each time an AR is written to, the LLW is incremented (modulo add). For example, instructions 1 and 3 in Figure 5b both write to AR 2, causing the LLW to hold the value 2. At the end of the trace, LLW points to the PR which holds that AR’s live-out value.

On trace completion, an AR’s GLW should be updated to its LLW register, so that subsequent traces can calculate the correct index for their live-in registers. Rather than copy the index value from the LLW to the GLW for every AR, we choose to have a ping-pong bit to interchangeably access these two index fields for alternate traces. For example, in Figure 5b, at the beginning of the first trace, R2’s ping-pong bit is cleared and the GLW referenced is 0. At the end of the trace, the LLW is index 2. If the trace completes successfully, R2’s ping-pong bit is set and henceforth index 2 is chosen to be the GLW. In case the trace fails, the ping-pong bit is unmodified and the trace restarts in program order, guaranteeing that the most recent value of the AR prior to mispredicted trace execution is read. Only OinO mode can flip an AR’s ping-pong bit.

Our design allows a maximum of 4 PRs per integer and floating point register, and up to 16 PRs for Conditional code registers in the ARM ISA. The constraint that an AR can be mapped to at most a fixed number of PRs forces the trace schedule generation algorithm on big to discard many schedules, limiting achievable energy savings. We investigate OinO’s sensitivity to this constraint in Section 5.

---

**Figure 5:** Handling false dependencies on OinO with a two-level register renaming scheme, where the Level-1 is done by big and encoded with the schedule and Level-2 is done on the OinO during execution.

### 3.2.2 Speculative Trace Start

Since little is capable of speculative execution, DynaMOS allows consecutive traces to start issuing after the previous memoized trace issues, without waiting for its completion. This is subject to the constraint that there are never more than 4 versions of an AR in the pipeline. DynaMOS achieves this by flipping the ping-pong bit on successful trace issue, instead of commit. In order to preserve correctness, a Commit–GLW register stores the last successfully committed index of an AR when a trace commits. In case of mispeculation, the ping-pong bit is flipped back, the Commit–GLW is copied to the GLW register, and InO mode takes over execution. This design constrains the number of active traces in the pipeline to at most 2, which is reasonable, since an InO core can’t get too far ahead without stalling on a use.

### 3.2.3 Handling Speculative Memory Operations

OoO cores reorder memory operations to maximize performance. However, executing loads/stores before older stores will lead to erroneous execution if they refer to the same address. The big uses a load/store queue (LSQ) to track in-flight values for the same memory address and detect memory aliasing. As OinO mode executes schedules with reordered memory operations, it must also handle memory aliasing. To this end, DynaMOS supplements little with a simplified LSQ.

However, without knowledge of the original program ordering between memory operations, the LSQ cannot function correctly. The beginning of every trace therefore also includes a fixed size meta-block to hold this information. This block, shown in Figure 6, contains the relative program sequence number for each memory operation with respect to the first memory operation of the trace. For example, while “Str b” is issued out of program order, its original relative sequence number (2) is recorded into the meta-block.

In the OinO mode, out-of-order memory operations are allocated entries in little’s LSQ by indexing into the structure by their original sequence numbers. Allocating memory operations in program order in the LSQ thus allows memory alias checks to be performed cor-
Figure 6: Memoized memory operations track their original ordering and are inserted into little's LSQ in program order.

3.2.4 Handling Precise Interrupts

An interrupt in InO mode, is treated as a misspeculation event, causing a pipeline and LSQ flush. The trace restarts in InO mode which handles the interrupt.

3.3 Trace Generation and Selection

To execute a memoized trace in InO mode, we must first have access to them. big is responsible for determining, building, and storing traces in DynaMOS.

Firstly, big determines trace boundaries, and calculates a unique TraceID for each trace. Nominally, a trace would include all instructions between two consecutive backward branches. However, it is occasionally necessary to truncate or extend traces due to length constraints. The trace length must be sufficiently long (more than 20 instructions), to allow useful reordering among instructions. Similarly, the trace length is bounded by big's instruction window size, this being the maximum window over which reordering can occur.

Next, memoizable traces need to be identified and stored. A trace's memoizability quotient is tracked by a 4-bit confidence counter in the Trace Selection Table (Figure 7). If the In-STC bit is set, blocks are fetched from the STC into InO's fetch buffer. Otherwise, they are fetched from the ICache and little mode executes.

A fill buffer similar to that used in [14] is used to combine instructions in issue-order on the big. This is validated at commit to ensure only non-speculative instructions within the traces are bundled together and stored into the STC. While writing to the fill buffer, the big encodes each instruction with a renamed register suffix (Section 3.2.1), increasing each register field by 2 bits. The first block of a trace in the STC is pointed to using a set-ID associated with each trace and all other blocks are accessed sequentially. As our traces are of variable lengths, they may span multiple blocks. A special kind of Trace marker denotes trace completion.

For capacity replacements in the STC, first, traces that have been rendered un-memoized due to highly varying schedules and then LRU traces are picked until sufficient space is created for a new trace. To avoid fragmentation of traces, we implement compaction in our STC, based on [16]. Since writes to this cache happen only on big commits, and are not on the critical path of InO's fetch, the overheads of compaction do not affect performance. Multiple writes to the STC might be necessary on a replacement, costing energy. But in our experiments with a 4kB STC, replacement happens rarely, amortizing the costs. The strict trace selection algorithm described previously ensures that only beneficial traces occupy the STC, resulting in high hit rates.

3.4 Controller

We adopt the controller of Padmanabha [5] to decide...
which core, big or little, should execute the current trace to maximize efficiency. The controller maximizes energy savings while keeping a constraint on the allowable performance loss (5% as compared to running only on the big). Traces with high ILP and/or MLP are run on big to maximize performance while traces with numerous dependent cache misses and/or branch mispredictions are run on little to gain energy savings.

The controller needs to make a two-level prediction, i.e., what trace will be seen next and which core should execute it. It uses a combination of history and program context to predict the upcoming trace. A traces preference to a core is determined by its relative performance on either core in the past. Measuring performance on the core it actually executed on is easy; a linear regression model (trained offline using SPEC’s train data set) is used to estimate its performance on the other core, similar to that in [5]. Performance metrics of a trace, such as cache misses, branch mispredicts, ILP, MLP, its memoizability quotient and estimated migration overhead, are fed as inputs to this performance model. Overall performance target is maintained by a feedback PI (Proportional-Integral) controller.

Unfortunately, while DynaMOS can detect memoizable traces that are, on average, 40 instructions long, the underlying architecture is incapable of transitioning between big and little at that granularity. Therefore, we are forced to combine consecutive traces together to form a super-trace, consisting of approximately 300 instructions. A super-trace is deemed memoizable if more than 80% of its instructions are present in the STC. We augment the super-trace prediction mechanism with this information, to give prediction accuracies of 80%.

3.5 Switching overheads
DynaMOS adopts the Composite Core [4] architecture, which tightly couples the big and little cores by enabling shared access to stateful structures, i.e., instruction and data caches, TLBs, and branch predictor. Only the register file has to be explicitly transferred between the cores on migration. This avoids the need to rebuild architectural state when migrating to the other core. Overhead for switching between cores is thus small, consisting of pipeline and store buffer drain of the active core, register file transfer and pipeline refill on the other, totaling 0.1% of execution time. These low overheads allow DynaMOS to switch rapidly between cores when memoized traces are encountered.

4. METHODOLOGY
We perform cycle-accurate simulations on the Gem5 simulator [17] to model performance of the cores and memory, the overheads of switching between the cores, and all the building blocks of DynaMOS. The high performing big core is a deeply pipelined 3-issue superscalar OoO processor with a large ROB and LSQ. Its ability to dynamically reorder instructions allows it to achieve 1.6x higher performance than little on an average for SPEC2006 benchmarks. The energy efficient little is an InO core that has the same superscalar width and FUs as big. This allows OinO to execute OoO schedules verbatim. The simpler hardware resources of an InO core allow it to consume a lower energy per instruction than the OoO core, saving 60% energy overall. Additionally, its shorter pipeline length affords it a quicker branch misprediction recovery. Table 1 describes the configurations used in detail. We used the McPAT modeling framework to estimate static and dynamic energy consumption of the core and L1 caches [18]. A core is assumed to be clock gated when inactive rather than power gated, as power gating adds significant overheads that do not scale at fine-granularity switching. The little pays an extra energy overhead for accessing the LSQ, bigger PRF, and STC when OinO mode is active.

For our evaluations, we use simpoints [19] from the SPEC2006 benchmark suite, compiled using gcc with -O2 optimizations for the ARM ISA. A maximum of 5 simpoints comprising of 1 million instructions each were picked from the first 2 billion dynamic instructions for each benchmark. Across 26 benchmarks (all those that compiled and ran on our simulator), we found 108 representative simpoints. Results for each simpoint are weighed and grouped with its parent benchmark.

5. RESULTS
This section provides quantitative benefits of DynaMOS and compares them to oracular knowledge and prior work [5, 10], along with intuitions as to why some benchmarks are more amenable to our technique than others.

5.1 Utilization of Little
The energy savings attainable by DynaMOS is proportional to the utilization of the little core. Utilization is defined as the fraction of the dynamic program executed on little, and is shown in Figure 8. DynaMOS tries to maximize this utilization while limiting the slowdown to 5% compared to big. We compare our results to an oracle and identically resourced Composite Core architecture without OinO mode provisions (CC). The oracle assumes perfect knowledge of execution and zero switching overheads, thus providing the most optimistic utilization of little given a performance constraint. It chooses a best schedule for memoizable traces over the entire application and runs that on the OinO mode. 50% of the application runs on little on average with oracle, of which 43% is executed on OinO mode.

We see that with practical constraints, DynaMOS is able to run 37% of its instructions on little, which is a

<table>
<thead>
<tr>
<th>Architectural Feature</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Big backend</strong></td>
<td>3 wide superscalar @ 2GHz</td>
</tr>
<tr>
<td></td>
<td>12 stage pipeline</td>
</tr>
<tr>
<td></td>
<td>128 entry ROB</td>
</tr>
<tr>
<td></td>
<td>180 entry integer register file</td>
</tr>
<tr>
<td></td>
<td>256 entry floating-point register file</td>
</tr>
<tr>
<td><strong>Little backend</strong></td>
<td>3 wide superscalar @ 2GHz</td>
</tr>
<tr>
<td></td>
<td>8 stage pipeline</td>
</tr>
<tr>
<td></td>
<td>180 entry integer register file</td>
</tr>
<tr>
<td></td>
<td>128 entry floating-point register file</td>
</tr>
<tr>
<td><strong>Memory System</strong></td>
<td>32 KB L1 iCache (Shared)</td>
</tr>
<tr>
<td></td>
<td>32 KB L1 dCache (Shared)</td>
</tr>
<tr>
<td></td>
<td>1 MB L2 Cache with stride prefetcher</td>
</tr>
<tr>
<td></td>
<td>2048MB Main Mem</td>
</tr>
</tbody>
</table>

Table 1: Experimental Core Parameters
2.9x increase over prior work (13%) [5]. This is due to OinO mode’s comparable performance to the big, allowing more instructions to run on little without incurring additional performance loss.

5.2 Benchmark analyses

DynaMOS excels for benchmarks like hmmer, bzip2 and h264ref that have high code locality, and recurse heavily in loops with straightforward control and data flow. Contrariwise, the inherent lack of memoizability in benchmarks like astar, gcc and gobmk hinders DynaMOS’ operation. However, little utilization is maintained because their low ILP and/or predictability allows them to run on InO mode.

5.2.1 Percentage of memoizable code

The amount of execution that can be memoized clearly dictates the profitability of DynaMOS. The oracle bar in Figure 8 shows how much of the execution can theoretically be memoized given a performance constraint (Note: Figure 2 in Section 2 shows the same result albeit with no performance constraint). Across the SPEC suite, we see benchmarks showing varied memoizable quotients. For example, the oracle schedules a meager 5% of execution on astar; astar implements a path finding algorithm that reiterates in the same function repeatedly, although dynamic instances take different paths. Hence, though there is code locality, there is no trace locality, disqualifying most of its execution from being memoized. On the other extreme, bzip2 goes through a dominating simpoint phase comprised of only 7 loops, of which 5 can be memoized, showing great potential for OinO execution.

5.2.2 Misspeculation overheads

Schedules that misspeculate have to incur a penalty of abort and replay, negating the benefits of running on OinO. Misspeculations occur when a forward branch in the trace takes the unexpected path or when a memory alias error is detected. Our trace selection algorithm heavily penalizes traces that abort often, so that they are disqualified from being scheduled on OinO, thus minimizing additional penalty cycles. Figure 9 shows the percentage of schedules that abort while in OinO. In gobmk, there can be many traces that spawn from a common header PC. Its high branch misprediction rates [13] indicate that these traces are not predictably repeatable. Hence, although each individual trace is highly memoizable (as was shown in Figure 2), the algorithm disqualifies all but 9% of them to run on OinO (Figure 8). Consequently, only 7.8% of the traces that are finally picked to go on OinO abort, adding a time penalty of 0.3% of total execution.

5.2.3 OinO’s architectural limitations

Another artifact of our architecture is the limitations on the amount of reordering achievable on OinO. Our schedules are constrained by the number of PRs an AR can be renamed to in the trace body, the number of memory operations allowed, and its size. Figure 10 tries to quantify these effects by inhibiting big in similar ways. The first bar shows big with a smaller instruc-
tion window of 40 instructions, the average trace-size observed empirically, limiting the number of instructions it can reorder over. Next, we limit the number of PRs it can access. We compare these to little with and without OnO mode enabled for the whole program.

On average, little with OnO attains 91% of a constrained big's performance. This shows that the 9% of performance loss can be attributed to the inherent lack of memoizable traces in the program, preventing OnO mode from being used. Note that little with OnO gains a speedup of 30% over InO mode, achieving 82% of big’s performance on average.

The oracle classifies 66% of execution as memoizable in cactusADM (Figure 8). DynaMOS fails to replicate this because the dominant simpoint in cactusADM comprises 3 regular traces that are more than 1000 instructions long. DynaMOS limits the maximum length of traces to 128 instructions and no reordering is allowed around trace boundaries. The unrestrained big can exploit more MLP and ILP by reordering over the entire trace. Figure 10 illustrates that with a limited instruction window big loses 9% performance, dropping further to 85% of unrestrained big with a smaller PRF. On the other extreme, hmmer, has a small dominant loop which has to be unrolled 4 times to reach our minimum trace length, requiring more than 4 PRs to rename to per AR, preventing it from being run on OnO.

DynaMOS’s optimization of speculatively starting a trace before the previous trace commits (Section 3.2.2), saves the OnO mode 1% performance.

5.3 Energy Savings

Figure 11 illustrates the energy conserved on DynaMOS as compared to running only on big. Overheads due to leakage from the inactive core (which is clock-gated) and those imposed by the CC controller[5] are included. Including OnO mode on little adds around 8% energy overhead to an InO, from accessing a bigger PRF and LSQ, but reduces its run-time significantly. The addition of a 4kB STC adds more leakage overheads, but contributes to lowering of OnO’s energy by providing instruction fetch access to a smaller cache. In case of mispeculation however, little faces an energy penalty of accessing instructions from both the caches. Overall, we observe that DynaMOS conserves up to 30% of big’s energy, a 2.1x increase from previous work.

5.4 Performance compared to Big

DynaMOS attempts to trade-off a user-configurable level of performance to maximize energy efficiency. This allowable performance loss is assumed to be 5% of big for the purpose of this work, which is maintained as shown in Figure 12. DynaMOS falls short of the target in a few benchmarks; mcf contains traces with loads that frequently miss in the caches. A schedule that has been created assuming a dCache hit latency for the load stalls on little when the load misses. The inability of OnO mode to react to such unexpected behavior and reorder instructions imposes performance loss. A high trace misprediction rate (Figure 9) also causes slowdown due to the need to rollback and replay. Note that lesie3d and bwaves overshoot the performance target, because the CC predictor overlooks some opportunities to switch to little. More fine tuning of this controller should mitigate this problem.

5.5 Sensitivity Analyses

For our work, the little is configured to be 3-wide superscalar so that memoized schedules from a 3-wide big can be replayed verbatim on OnO. However, this is not essential for correct execution of DynaMOS. Figure 13 illustrates the efficacy of DynaMOS on differently configured systems - particularly with a bigger big and a smaller little. All resources in the cores, including FUs are scaled appropriately. The results are shown relative to the corresponding big in that system. As per our current implementation, if an issue slot in a trace is wider than the issue width of little, it simply splits each slot into 2 or more slots. For example, a 2-wide little splits an issue slot from a 3-wide big into...
5.5.2 Sensitivity to Size of Schedule Trace-Cache

The STC size dictates how many schedules are available to InO when execution migrates to little. Figure 14 illustrates the effects of little’s utilization and performance as the STC size varies. As expected, utilization increases as the size increases from 1kB to 4kB because of reduction in capacity misses. Beyond that we observe that utilization plateaus out as most of the required schedules fit in the cache, prompting us to choose a 4kB STC for DynaMOS.

5.6 Comparison to Execution Cache

Research has proposed to conserve energy of the pipeline front-end by caching recurring traces to exploit temporal locality of instructions. We compare DynaMOS to two of the most relevant works in Figure 15. Loop Caches (LC) [20] cache encoded or decoded instructions from recurring loops, thus eliminating energy consumed by the fetch and/or decode stages. Execution Caches [10] exploit memoizability in schedules and conserves energy by bypassing the fetch, decode, rename and issue stages of an OoO pipeline and sending instructions directly to execution. Figure 15 shows that on average adding a 4kB LC to big conserves 4% energy while adding a 4kB EC saves 23% of energy at the cost of 16% loss in performance. Adding a feedback controller similar to that in DynaMOS can constrain performance loss to 5% by limiting the use of EC when performance degradation is observed, subsequently reducing energy savings further to 10% (Big+EC@5% bar). Two reasons contribute to DynaMOS’s relative benefit – First, it moves execution over to a different InO core which runs at 30% of big’s power when traces are memoizable, while power hungry structures like the ROB, LSQ are active throughout execution in a big with an EC. Second, the InO mode also executes non-memoizable but low performance traces at minimal performance loss. astar, for example, has a low memoizable quotient, and hence schedules are rarely available in the Trace-cache or EC. Still, DynaMOS runs 18% of its code on little in InO mode, yielding 20% more energy savings.

5.7 Energy and Area Overheads

Little is provided with a larger PRF to ensure seamless and inexpensive register renaming. In our implementation it has 128 entries, 4x the existing size. Each AR needs 2 bits each for Commited, Global and Local LWs fields, and 1 ping-pong bit, totaling 7 bits of information with each AR (Section 3.2.1). Little also includes a 32 entry LSQ, increasing its energy by 8%. These bigger structures can be clock-gated when running in InO mode. Adding a 4kB STC adds 0.11mm² of area and 10% leakage power overhead to the little. We do not however model the energy overheads of the logic added. The inactive pipeline, which is clock-gated, contributes to the leakage energy overhead. Since our architecture is targeted toward low-power cores, we use low-leakage technology with low operating power, minimizing this overhead to <4%. Prior work [5] estimates the controller to incur overheads of 1.875kB storage and 0.033W power, while the performance estimation model covers 0.02mm² area, and consumes <5mW power.

6. RELATED WORKS

6.1 Efficiency through Heterogeneity

Researchers have explored the architecture design space in various dimensions of heterogeneity. These include cores with different sets of microarchitectural parameters [21, 22], specialized hardware for specific codes [23, 24], and heterogeneous ISAs [25]. Commercial products include ARM’s big.LITTLE [1] and NVidia’s Kal-El [26] which combine high and low performance general purpose cores, and IBM’s Cell [27] and Intel’s vPro [28] which run special code on customized elements.

The architecture of the heterogeneous multicore dictates the granularity of application migration amongst
them. Higher switching costs of coarse-grained heterogeneous systems [1, 2] enforce switching granularities of the order of milli-seconds. Novel architectures minimize migration costs by either sharing of structures between cores [4, 7] or reducing the distance between them using 3D technology [6]. Composite Cores [4] shares access to L1 caches, TLBs, fetch unit and branch predictor between an OoO and InO core.

Another approach to heterogeneity is to reduce the voltage and frequency of the core (DVFS) to improve the core’s energy efficiency at the expense of performance [29, 30]. Recent work [31] shows that microarchitectural heterogeneity, which can altogether eliminate use of energy-intensive structures like the ROB and issue logic helps conserve more energy than DVFS. Nevertheless, DVFS is a complementary technique that can be applied to DynaMOS for additional energy savings.

6.2 Trace-Based Execution

Several prior works proposed storing instructions in logical dependency order rather than the order in which they are stored in memory [14]. They aim to improve efficiency of fetch by grouping instructions that are likely to execute together as a trace in a trace cache.

Compilers use profile-based static scheduling mechanisms [32] or run-time binary optimization [33] to create optimized instruction schedules. Most compilers however, cannot schedule beyond a few basic blocks due to data and control flow unpredictability. Allowing OoO hardware to create the schedule ensures that InO is provided with the best-case schedule for a particular trace for a particular dynamic phase in the program. Software approaches, e.g. Nvidia’s Project Denver, dynamically re-compile recurring traces to create high-performing schedules for an InO core. Compared to hardware implementations, such methods react tardily to finer-grained phase changes in applications. Also, re-compilation of traces cannot be done indiscriminately, while DynaMOS modifies schedules as needed.

Industry introduced trace caches as a method to improve performance and energy-efficiency by caching post-decode traces, thus allowing expensive CISC decoders to be turned off [9, 34]. Section 5 compares DynaMOS’s behavior to relevant schemes that exploit temporal locality of instructions to conserve pipeline front-end energy. Villavieja et.al [35] propose to transition from an OoO core to a VLIW core with all front-end stages turned off when a memoized VLIW schedule is available. In case of highly unpredictable branches, the lack of memoizability forces these works to fall-back on the OoO core. DynaMOS on the other hand, can still gain energy-efficient execution on the InO mode.

Other works propose using two pipelines: one for trace generation and another for trace execution. Turboscalar [12] has a thin cold pipeline that discovers ILP over a long instruction window and a fat hot pipeline to exploit this knowledge. Perhaps the work that most closely resembles ours is DIF [11]. They use a primary OoO engine to create and cache schedules, which are then formatted as a VLIW and made accessible to a secondary VLIW-style accelerator. DIF’s aim is to increase performance by dynamically increasing the ILP exposed to a narrower superscalar OoO and running it on a wide VLIW machine. Our goal is to trade-off performance for energy-efficiency by exposing ILP to a simpler InO core, allowing the complex OoO to remain idle. Also, our little doesn’t rely exclusively on big to provide it with instructions, and is capable of working stand-alone when energy-efficiency is of utmost importance. We leverage two-level renaming that was proposed in this paper, and modify it so that we do not have to generate and store live-out register maps for every trace, which is expensive to do dynamically. The HBA [7] architecture comprises of a common front-end and register file which feeds into one of either OoO, InO, and VLIW backends. It stores instruction schedules at the granularity of a couple of basic blocks and heuristically decides to run on one of the backends.

7. CONCLUSION

In this paper, we observe that the sequence in which instructions in an OoO pipeline are issued to execution tends to remain similar for repeating traces. OoO cores perform redundant work by recreating the same schedule for every instance of the trace. Based on this insight, we provision an InO core with the ability to read and execute issue schedules recorded by an OoO core. We observe that with equal resources, an InO core can nearly achieve OoO’s performance, at a fraction of the energy cost. We propose the DynaMOS architecture, which consists of a tightly-coupled big and little core, wherein execution migrates with low overheads from big to little for traces with either low IPC or with memoized schedules. To this end, little is provisioned with an OoO mode which can execute memoized out-of-order schedules while guaranteeing correctness. DynaMOS was shown to increase little’s utilization by 2.9x over prior work, thus saving 32% energy over execution on only big, with an allowable 5% performance loss.

8. ACKNOWLEDGEMENTS

This work is supported in part by ARM Ltd and by the NSF under grant SHF-1217917. The authors would like to thank fellow members of the CCCP research group, and the anonymous reviewers for their time, suggestions, and valuable feedback.

9. REFERENCES


