Design Trade-offs for Memory Level Parallelism on an Asymmetric Multicore System

George Patsilaras, Niket K. Choudhary, James Tuck

Design Trade-offs for Memory Level Parallelism on an Asymmetric Multicore System

George Patsilaras, Niket K. Choudhary, James Tuck
Department of Electrical and Computer Engineering
North Carolina State University
{gpatsil, nkchoudh, jtuck}@ncsu.edu

Abstract

Asymmetric Multicore Processors (AMP) offer a unique opportunity to integrate many kinds of cores together with each core optimized for different uses. However, the impact of techniques for exploiting high Memory Level Parallelism (MLP) on core specialization and selection on AMPs has not been investigated. Extracting high memory-level parallelism is essential to tolerate long memory latencies, and such techniques are critical for speeding up single-threaded codes which are memory bound. In this work, we explored multiple core configurations with different widths and frequencies and concluded that a narrow faster core is better than a wide slower core for regions of high MLP.

We use an effective hardware-level scheduling mechanism, which requires identifying MLP phases on the fly and scheduling execution on the appropriate core. We successfully exploit the custom MLP core during clustered L2 misses and otherwise use the wider issue core. Compared to a single-core design optimized for both modes of operation, our AMP design provides a geometric mean performance improvement of 4% and 10% for SPECint and SPECfp, respectively, with a maximum speedup of 19.5%. For the same study, it achieves a 10% and 25% energy delay^2 reduction fo SPECint and SPECfp, respectively.

1 Introduction

Asymmetric Multicore Processors (AMPs) have been proposed as a means to achieve an improved performance per watt ratio for a wide range of applications [14, 15, 16], when compared to Symmetric Multicore Processor Systems. This advantage comes from the way AMPs can exploit diverse behavior across and within applications. Applications targeted by AMPs tend to be split into two groups: single-threaded sequential applications or parallel applications. A further classification of single-threaded applications is: CPU-intensive applications, which have high amounts of instruction level parallelism (ILP), and Memory-intensive, which have high cache-miss rates and thus high amounts of processor stall time.

AMPs have been shown to be beneficial when executing the CPU-intensive single-threaded applications on the powerful cores [1] while being more power efficient for the Memory-intensive single threaded applications by executing on the simpler cores during regions of high L2 cache miss rates [14]. Finally, AMPs provide higher performance per watt when executing highly parallel applications on a group of simpler cores [16].

Despite being in the multi-core era, sequential performance matters. Memory level parallelism (MLP) has been proposed as a way to boost the performance of applications that stall frequently. Rather than waiting for one access at a time, the goal is to exploit available memory bandwidth to request many memory accesses at once. A variety of hardware-only MLP enhancing techniques have been proposed [6, 18, 9, 19, 8, 3, 13, 31, 17, 27, 30]. These techniques are advantageous since they can transparently accelerate sequential codes, but they are limited by their high energy consumption.

Some of these MLP techniques leverage precomputation to issue loads in advance. Other techniques leverage hardware within a single core to detect a long latency load then speculate past it to generate overlapping misses [9, 3, 13, 31]. Multithreaded approaches have been considered which automatically construct prefetching threads[9], or tightly couple two cores to act as a large instruction window [30]. Given the importance of tolerating long latencies to memory *within a single thread*, future processors will likely incorporate techniques to overlap long latency misses.

Given the performance benefits of techniques for boosting MLP, a key question is *how to best integrate high MLP techniques on an AMP*. The possible answers to this question are not trivial or obvious. For example, applications which benefit the most from MLP techniques tend to have a low IPC and favor simpler cores for power-efficient performance, thereby favoring the integration of MLP techniques on such cores. But, when MLP techniques are added to the cores, past studies have shown that wider cores are needed to extract enough cache misses to make high MLP techniques worthwhile [4]. On the other hand, smaller cores can run faster with better power efficiency than larger cores.

Therefore, core selection for MLP will depend on the behavioral characteristics of applications with problematic L2 miss rates, the core-level needs of MLP techniques, and the design implications of different core widths.

This article makes the following contributions:

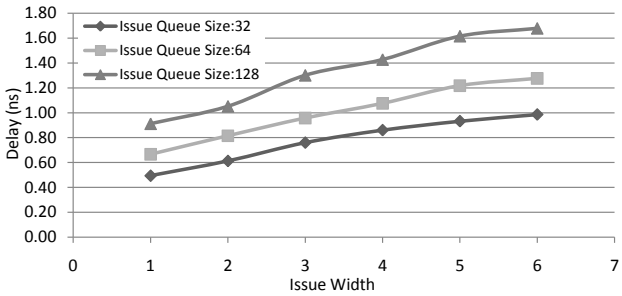
- We are the first to design an AMP which couples an independent ILP core with a customized core, that incorporates an MLP technique.
- We explore the customization of our MLP technique by investigating different combinations of core designs for code regions with varied L2 miss rates. With this analysis, we can identify trends in core behavior and pinpoint a better core design among those studied for our MLP technique.
- For Checkpointed L2 Miss Processing with Value Prediction (*CLP+VP*), a scheme similar to CAVA [3] and Clear [13], we find that a narrower 2-wide issue core outperforms a 4-wide issue core. This advantage is born from the interaction of higher frequency and the systematic behavior of *CLP+VP*. Overall, our analysis advocates an Asymmetric Multicore Processor design that supports MLP by integrating *CLP+VP* on the 2-wide issue core.
- We propose Symbiotic Core Execution (SCE) to exploit fine grained differences in application behavior to run moderate to high MLP regions on the narrower core customized for MLP and the other regions on an aggressive 4-wide issue core designed for high ILP.
- For SCE, we identify an effective scheduling mechanism which judiciously switches cores to exploit regions of high MLP on the customized MLP core without incurring too much overhead from switching.

The rest of the paper is organized as follows. Section 2 gives an overview of AMPs and the MLP enhancing technique we consider; Section 3 presents a detailed study of different core designs for varying levels of MLP. Section 4 describes our proposed Symbiotic Core Execution; Section 5 discusses our methodology and provides a detailed evaluation. Section 6 is devoted to related work, and Section 7 concludes.

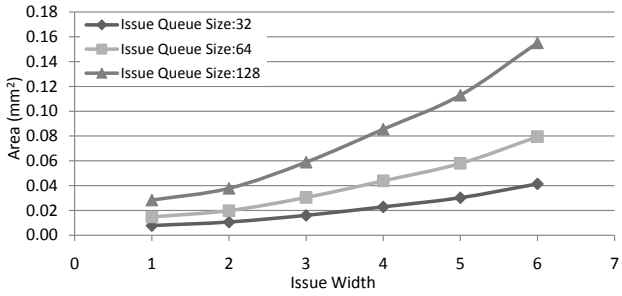
2 Background

2.1 Asymmetric Multicore Processors with Core Customization

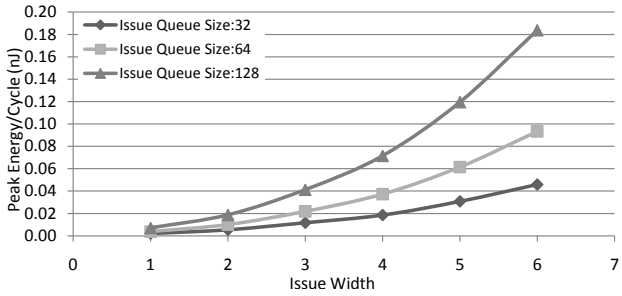
AMP customization works by tailoring a core for the particular needs of an application or workload. For example, some applications benefit tremendously from variations in branch predictor design, cache size, or issue window size [15, 20] that would not be generally applicable to a wide range of programs. By building many customized cores



(a) Delay Scaling



(b) Area Scaling



(c) Peak Power Scaling

Figure 1: Scaling of Issue Queue in terms of Delay, Area, and Peak Power

that target such behaviors, applications can benefit more by running on the customized core than running on a core designed for the general case.

Designing customized cores requires a careful design space and design cost exploration. A processor design has an associated cost, where the cost can be quantified in terms of propagation delay, power consumption, die area, design effort, manufacturability, or fault vulnerability. A complex microarchitecture might enhance IPC, but at the same time could increase the propagation delay. For instance, increasing the size of the issue window and issue width can boost IPC for applications with abundant ILP, but at the same time, clock rate may decrease to accommodate the larger content addressable memory and deeper select tree.

Fabscalar[5], is a state of the art tool that enables architects to synthesize customized designs and evaluate the effects of different designs, in great detail, in terms of frequency, area and power. Using fabscalar we can synthesize a Verilog model of an arbitrary superscalar processor and

analyze how sizing structures can affect frequency. Figures 1(a) 1(b) 1(c) show the impact of increasing the issue window on the delay, area, and peak energy consumption of the wakeup-select logic for different issue widths. These assume a 45nm technology and the same input voltage. As we can see from the graph the smaller the width and issue queue size the faster the clock frequency can be.

We used Fabscalar to perform a design space exploration that searched for the fastest 1-wide, 2-wide, and 4-wide issue cores. In our search, we assumed a pipelined architecture with a constant depth and fixed supply voltage, then we varied the issue width and all related microarchitectural structures. We found that the 4-wide cores maximum frequency was 3GHz, the 2-wide core frequency was 3.6GHz and the 1-wide core had a maximum frequency of 4.5GHz (more architectural details can be found in Section 5.1). The design search considers all possible timing critical paths of a modern superscalar out-of-order processor, for example wake-select logic, rename logic, cache access time etc. [21], to synthesize a processor with a realistic clock frequency. Since we keep the total pipeline depth constant for our exploration, the synthesized core reflects the trade-off between the pipeline complexity and the propagation delay.

As in any design space exploration, it is important to search an appropriate set of designs. Fabscalar considers many circuit-level and architecture-level optimizations, although it is not exhaustive. Therefore, a design team could find other core designs our search did not consider. In general, however, attempts to increase the frequency through microarchitectural complexity do not always have the expected effect on performance, area, or power consumption. For example, while pipelining can help mitigate the increased propagation delay for larger structures or wider processors, it can also lead to a decreased IPC [12, 2]. While our exploration is not exhaustive, we believe the relative performance, while not absolutely the same, may be indicative of what a state-of-the-art design team could achieve in industry.

In the rest of the article, we use the core designs found by Fabscalar to investigate how to customize an AMP for our MLP technique.

2.2 Checkpointed L2 Miss Processing with Value Prediction: *CLP+VP*

Several MLP mechanisms have been described in previous work, such as Runahead Execution [19], CAVA [3], and others [13, 17]. Our basic mechanism, shown in Figure 2(a) adopts some features from CAVA [3] and Clear [13]. Once an L2 miss reaches the head of the ROB, a checkpoint of the register file is recorded at the point just before the load. Then, we place a predicted value in the destination register of the load and continue execution as a speculative epoch. By retiring the load with a speculative value, forward progress can be made while waiting for the memory operation to complete. Once the L2 miss completes, the processor checks the actual value with the predicted one; and, if the prediction was correct, we exit the speculative epoch. If the load's value was incorrect, execution is restored to the checkpoint, the value predictor tables are updated, and the processor resumes from the mispredicted load instruction. In order to avoid recovery in the case of a successful prediction, the L1 cache buffers the speculative state [3]. We will refer to this basic strategy as Checkpointed L2 Miss Processing with Value Prediction(*CLP+VP*). *CLP+VP* supports making predictions on multiple outstanding loads, but, only one checkpoint is kept, so any misprediction results in rolling back the entire speculative epoch back to the first predicted load value. Therefore, we adopt CAVA's mechanisms to prevent aggressive speculative execution through loads which are value predicted with low confidence.

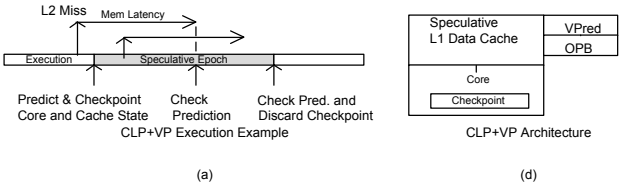


Figure 2: MLP regions

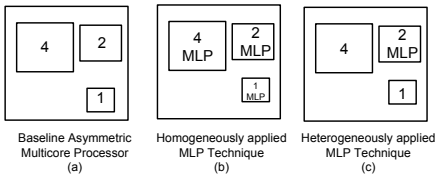


Figure 3: AMP with MLP Design Space

Figure 2(b) shows the *CLP+VP* architecture with the required Speculative Data Cache [3]. In addition, the OPB is the Outstanding Prediction Buffer. It tracks all outstanding predictions, and if the prediction does not match, the Recovery and Rollback logic is triggered.

3 MLP Design Trade-offs on an AMP

In this section, we investigate how best to integrate high MLP techniques in an AMP. As a motivational discussion, consider the systems shown in Figure 3. Figure 3(a) shows a Asymmetric CMP similar to that proposed by Kumar *et al.* [14] but we assume that a base core design is optimized for different issue-widths. We are primarily interested in answering two related questions. (1) is there a core preference when implementing an MLP technique? In other words, are MLP techniques heterogeneous and should be integrated on particular cores, or should we integrate the technique throughout the entire chip (homogeneously). Integrating a technique on every core is preferable only when it offers compelling advantages, given design effort and validation costs. Figure 3(b) shows the case where all cores receive the high MLP technique.

To answer these questions, we divide the analysis into

Label	Configuration
Core4	4-wide 3GHz Core
Core2	2-wide 3.6GHz Core
Core1	1-wide 4.5GHz Core
Core2 CLP+VP	2-wide 3.6GHz Core with CLP+VP
Core4 CLP+VP	4-wide 3GHz Core with CLP+VP
SCE CLP+VP	4-wide 3GHz Core+2-wide 3.6GHz CLP+VP core

Table 1: Asymmetric Core Configurations

two parts in the following sections. First, we investigate core preference for regions of code with a large L2 miss rate. This establishes baseline performance for the cores considered in our study. For every configuration looked at we used Fabscalar [5] for a logic synthesis of the entire processor in order to measure frequencies and power, we then use SESC [24] for our simulations. The core configurations are described in Table 1 however more details on our experimental setup can be found in Section 5.1.

3.1 Exploring MLP Potential on Different Core Widths

In this section, we investigate the behavior of our baseline AMP design in code regions sensitive to MLP techniques. Our goal is to establish a relationship between core performance on a code region and suitability for exploiting MLP in the same region. To establish enough data points to draw a strong conclusion about potential for MLP and core preference, we extract code regions of 10K instructions from a dynamic trace of the SPEC CPU 2000 applications and bin these regions according to their L2 miss rate. Hence, for a wide range of applications, we can average out behavior based solely on potential for MLP.

Figure 4 plots the performance of the *Core2* and *Core1* designs normalized to the *Core4* design. The dashed lines are forcing all cores to run at the same frequency, while solid lines are *taking frequency explicitly into account*. Each point on the y-axis is calculated by summing the execution time of all code regions for the L2 miss count specified on the x-axis as measured on each core. This execution time is then used to calculate speedup relative to *Core4*. As a result each bin on the x-axis indicates a different L2 miss rate (L2 misses/10K instructions). In this analysis, we presume that regions with higher miss rates will have more *potential* for MLP. Since these cores have no additional MLP technique, they are not exploiting the MLP available from out-of-order execution.

The clear advantage goes to *Core4* across the entire MLP continuum under the same frequency. The relative performance gap is larger for low MLP and narrows significantly at higher MLP potential. This narrowing is the result of significantly less ILP in regions of many L2 misses, thereby eliminating *Core4*'s primary advantage.

When considering frequency, rather than maintaining a significant advantage across the continuum, *Core4* loses its advantage over *Core2* for L2 miss rates above 10 per region. Furthermore, *Core4* is no longer competitive with *Core2* for high miss rates above 90 per region and not

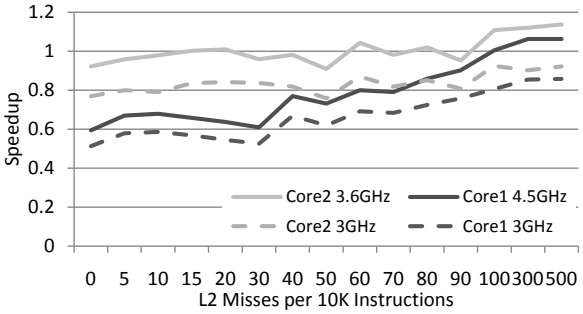


Figure 4: Performance versus MLP potential normalized to *Core4*

competitive with *Core1* above 100 misses. Since the frequent L2 misses prevent the wider core from leveraging its greater width for more ILP, the higher frequency cores are able to process instructions faster. However, the frequency advantage of *Core1* is not enough to overcome the higher IPC of *Core2*; hence, *Core2* always outperforms *Core1*. In between 10 misses and 90 misses per region, there is no clear winner when comparing *Core4* and *Core2* but we will consider *Core2* winner due to power savings of a smaller core.

This analysis is important because it shows that the narrower cores are not only better for power, as previously reported [14], but can also provide a performance advantage when designed and clocked at their highest frequency. Even though our design space exploration isn't perfect, given the same design team, a 2-wide core will likely be faster than a 4-wide one. Also, it is clear that core choice is related to MLP potential. Where there is little to no potential for MLP, *Core4* is undoubtedly the better choice even with a slower frequency. However, with moderate to high potential for MLP *Core2* is best.

3.2 Exploring *CLP+VP* On Different Core Widths

We analyze the performance when *CLP+VP* is added to each core in the AMP and evaluated it with respect to MLP potential. In Figure 5, clearly, *Core4+CLP+VP* is better than *Core4* with as few as 5 misses per region. We can also observe that *Core2+CLP+VP* is competitive with *Core4+CLP+VP* somewhere between 5 and 10 misses. From that point on, the designs appear to be equivalent with *Core2+CLP+VP* achieving a distinct advantage with more than 80 misses. *CLP+VP* eliminates many costly pipeline stalls due to L2 cache misses, however, high ILP still cannot be achieved because the MLP technique does not kick in until the L2 miss has been detected. This will delay the processing of the load enough to favor narrower cores with higher frequencies.

4 Symbiotic Core Execution

An AMP design that dynamically leverages the best core for a region of execution may be advantageous compared to

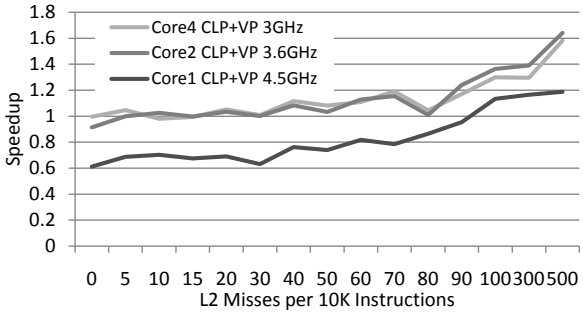


Figure 5: Performance versus MLP potential normalized to *Core4*

an application level policy which waters down the advantage of any particular core over the regions where it is less effective. We propose incorporating the *Core2+CLP+VP* core along with *Core4* on an AMP and leveraging fine grained scheduling in hardware to instantaneously choose the best core depending on the MLP potential present in the code. When a MLP technique is needed, execution switches to *Core2+CLP+VP*. When an application is not in a moderate to high MLP region, it executes on the *Core4*. We call our proposal *Symbiotic Core Execution* (SCE).

The term symbiosis is borrowed from biology: two self sufficient organisms exist symbiotically when they survive better together than when alone. Symbiosis is a compelling description because it identifies the cores as being independent and capable of working alone, as is often needed in a multiprocessing environment. However, when prudent, they can be used together for a greater performance advantage. A key challenge of SCE is efficient scheduling.

4.1 Effective Hardware Scheduling on the MLP Core

To effectively schedule for MLP, we build our policy around the observation that L2 misses tend to cluster. Hence, it is our goal to judiciously schedule work on the MLP core during regions of many L2 misses and switch back when the region ends. An effective hardware scheduling policy requires a balance between switching eagerly to ensure that no region is missed and switching lazily so that core switching is not invoked on isolated L2 misses nor incurs significant overhead.

4.1.1 Eager Switching for Clustered Misses

The first goal of our scheduling policy is to eagerly switch to the MLP core to exploit regions of clustered misses. We leverage our analysis from Section 3 to determine the rate of L2 misses needed for the MLP core to overtake the ILP core. According to Figure 5, on average, we see the cross over point at 10 L2 misses per 10K instructions. So, as a heuristic, we assume that we need to observe misses at that rate for the MLP core to be profitable. We identify this rate as r_{miss}. We use r_{miss} to identify the minimum number of L2 misses that should be observed in

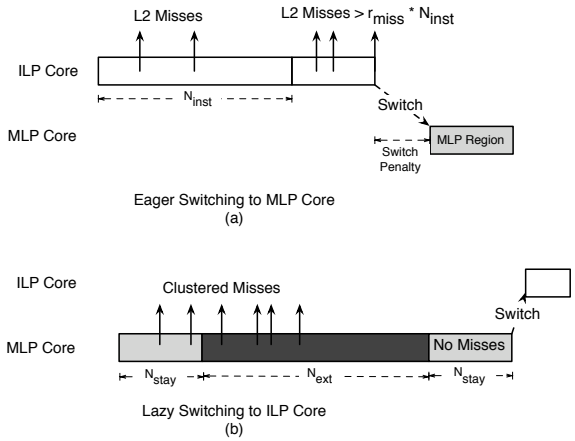


Figure 6: SCE Scheduling Illustration

a region of N_{inst} instructions in order to switch cores.

For each contiguous chunk of N_{inst} instructions, our scheduler counts the number of misses. If fewer than $N_{inst} \times r_{miss}$ misses are observed at the end of the region, the counter is cleared. As soon as $N_{inst} \times r_{miss}$ misses are observed, the application is switched to the MLP core. Figure 6(a) illustrates this policy.

A key challenge is tuning N_{inst}. If it is too small, we will switch every time we see an L2 miss. However, if N_{inst} is too large, it increases the likelihood that we wait too long to switch and miss good scheduling opportunities. After a series of experiments we concluded that $N_{inst} = 3000$ works well for a variety of applications. Assuming $r_{miss} = 0.001$, this means we are looking for 3 or more misses in a region to switch to the MLP core. The hardware cost associated with these counters is very small and similar mechanisms are already present in current microprocessors. Note that this miss rate corresponds to the point in Figure 5 when *Core2+CLP+VP* becomes the desirable core.

4.1.2 Lazy Switching to Correct Core

Once running on the MLP core, our scheduler evaluates a) if it was a correct decision to switch and b) when there is no longer a benefit from MLP. It is important to remain on *Core2+CLP+VP* long enough to exploit any MLP. However, if the decision to switch was erroneous, it is desirable to catch this mistake quickly before too much time elapses.

Since L2 misses cluster, if the application is entering a region of clustered misses, we expect an elevated miss rate If this phenomenon is not observed, it is unlikely there will be any benefit from the MLP technique. Therefore, we define N_{stay} as the number of instructions that must be executed on the MLP core; during that region, an L2 miss must be observed. If no misses are observed, it is likely that clustering is not present (or no longer present), and execution should return to the wide core. However, if a single miss is observed, we remain on the core for an extended execution, N_{ext}. At the end of the extended region, we perform the same evaluation again. Figure 6(b) illustrates

Frequency	3 GHz	3.6 GHz	4.5 GHz
Fetch Rate	6	4	2
Issue Rate	4	2	1
Retire Rate	4	2	1
ROB	128	96	64
LD/ST Queue	54/38	42/38	36/24
Mem/Int/Fp Units	2/3/2	2/2/2	1/1/1
Area(mm^2)	2.513	1.251	0.540
Pipeline: 3-cycle fetch, 1-cycle decode,1-cycle Rename, 1-cycle dispatch, 2-cycle issue,2-cycle issue,1-cycle RegRead, 2-cycle Mem/Int/Fp Units, 1-cycle DataCache, 1-cycle writeback, 1-cycle retire			
I L1 Cache: Size=32KB; Assoc=4-way; Line size=64B; RT=2 cycles Private D L1 Cache: Size=32KB; Assoc=4-way; Line size=64B; RT=2 cycles L2 Cache: Size=2MB; Assoc=8-way; Line size=64B; RT=10 cycles Main Memory RT=100 nanosec, Core switching delay=100 cycles BTB: size=2K; Assoc=2-way			
Branch predictor: bimodal size=16K; gshare-11 size=16K; Branch Mispred. Pen.=14 cycles H/W Pref.: 16-stream stride prefetcher; hit delay=8 cycles; buffer size= 16KB SCE param.: N_{stay}=700cycles, r_{miss}=0.001 and N_{exit}=3000cycles			
CLP+VP configuration: OPB = 128 entries Max. Outs. preds./Instr.=512/3072 BHLV+GLV predictor, table size=4096 entries each			

Table 2: Core details. Cycle counts are in processor cycles

this policy. We found that $N_{inst} = 3000$, $N_{stay} = 700$ and $N_{ext} = 3000$ work well.

4.2 Operating System Interaction

OS view of cores. We assume that the chip can operate in two modes. In SCE mode, we assume that the OS sees one logical core and hardware can initiate context switching among the symbiotic cores. In the case where two threads are waiting to execute, SCE is disabled, and the OS has a view of two available cores. Scheduling is done in a similar way as [16].

Context Switching. During SCE mode, when hardware determines the need to switch cores, it stalls the fetch engine. Once the pipeline is empty and all instructions from the pipeline retire, we context switch between the cores. We assume a constant delay of 100 cycles to model the latency for copying registers from one core to the other. Once the context switch is finished, we resume execution on the new core. Our simulations modeled behaviors such as cache warm-up penalty, effects on the TLB, and branch predictor due to switching cores. These costs are modeled separately from the 100 cycle register copy penalty.

5 Evaluation of SCE

5.1 Experimental Setup

To evaluate our AMP proposal we used SESC [24] an execution-driven simulator, and compiled SPEC2000 applications using a MIPS cross compiler built from GCC 4.4 [11]. We also used the FabScalar framework [5] to weigh the cost of different superscalar designs in terms of clock period, area, and power (static and dynamic). The FabScalar framework can be used to synthesize Verilog models of arbitrary superscalar processors, where each superscalar processor can be customized in terms of pipeline ways (width of the processor) and sizes of the memory structures within a stage.

Since a superscalar processor makes use of many

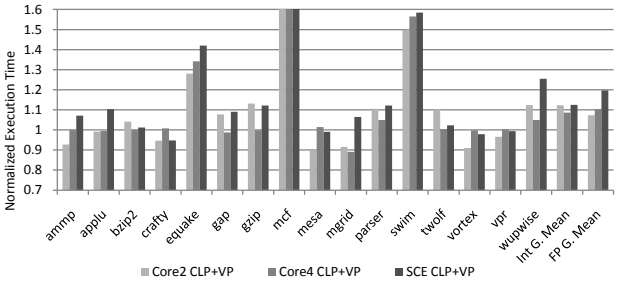


Figure 7: Performance comparison Normalized to Core

specialized and highly-ported RAMs/CAMs/FIFOs (e.g., physical register file, rename map table, issue queue, load-store queue, active-list etc.), we are also using the register file compiler from the FabScalar framework. The register file compiler uses custom layouts of multi-ported bit-cells and peripheral circuits to generate memory structures and characterize their access times and power consumption by doing SPICE level simulation.

We used Synopsis Design Compiler C2005.09-SP3 and placed-and-routed with Cadence SoC Encounter V7.1, using the FreePDK OpenAccess 45nm Standard Cell Library [28] to synthesize our different designs in order to estimate timing.

Table 2 shows the SESC configuration parameters used for the *Core4*, *Core2*, *Core1*, and *CLP+VP* configurations. Labels used in the graphs are explained in Table 1.

5.2 Performance

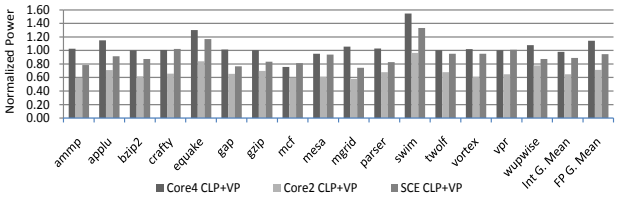
Figure 7 shows the speedups of *CLP+VP* described in Section 2.2 over *Core4*, our base case. We see that SCE with *CLP+VP* delivers a geometric mean speedup of 1.13 for SPECint applications and 1.20 for SPECfp over *Core4*.

Looking at individual benchmarks we see that *applu, equake, mcf, mgrid, swim,* and *wupwise* benefit from SCE utilizing the *CLP+VP* core for MLP regions and the high ILP core for the other regions. SCE scheduling yields higher performance than any of the cores on their own. For benchmarks with no benefits we see that they are characterized by low L2 miss rates.Finally, *crafty* and *vortex*, which have a slight degradation do not benefit from CLP+VP technique indicating that these benchmarks have no exploitable MLP regions.

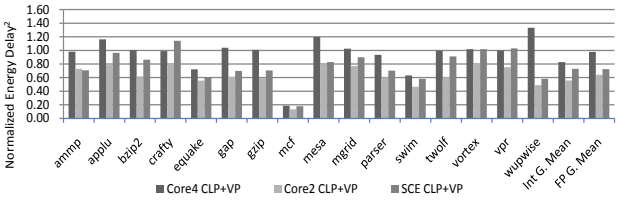
When comparing to a single core with the CLP+VP technique, we see that our technique is better. *SCE CLP+VP*'s speedup of 1.13 and 1.20 for SPECint and SPECfp applications, respectively, are higher than CLP+VP's, 1.09 and 1.10. The benefits come from the faster 2-wide CLP+VP core achieving better performance than the 4-wide CLP+VP for high MLP regions.

5.3 Power and Energy Delay[2]

Figure 8(a) shows the power consumption for each design we evaluated. This includes the static and dynamic power for the occupied core and the static power from the



(a) Power Normalized to Core4



(b) Energy Delay^2 Normalized to Core4

Figure 8: Power and Energy Delay^2 for Core4

core not used. Caches remain active for the entire execution, and so we modeled the static and dynamic power during the entire execution for regular accesses and invalidations triggered by the coherence protocol. As we can see, our *SCE CLP+VP* design consumes less power compared to *Core4+CLP+VP* by a geometric mean of 9% and 19% for SPECint and SPECfp, respectively. SCE however, does require more power than *Core2+CLP+VP* but this is a direct consequence of using *Core4* to accelerate non MLP regions.

Overall, SCE is power efficient despite the added MLP core. The main reason is because the L2 Cache constitutes more than half the total chip area, and thus contributes the most to the total static and dynamic power consumed. The second reason is that the dynamic power consumed by the 4-wide core is significantly higher than the dynamic power consumed by a 2-wide core. During regions where execution is on the MLP core the dynamic power difference compensates for the extra static power added for the MLP core.

Figure 8(b) displays the energy delay^2 for each configuration. Overall, SCE with CLP+VP reduces the energy delay^2 by 27% and 28% for SPECint and SPECfp over *Core4*. When compared to the *Core4+CLP+VP* design, it is reduced by 10% and 25% for SPECint and SPECfp, respectively. Note that *equake* and *swim* are particular advantageous when considering energy delay^2 because their higher performance offsets their higher power. On the other hand, we see that *crafty, vortex,* and *vpr*, which do not benefit from CLP+VP in performance, have a slightly worse energy delay^2 over *Core4* due to the added core's static power and the added instructions executed.

5.4 Switching Overhead

Table 3 displays details on the switching overhead for our SCE proposal. The *Overhead* column indicates the overhead due to switching cores over the total execution.

Benchmark	Total Overhead	StallCore4	StallCLP+VP	Switch
ammp	2.02%	61.81%	15.56%	22.63%
applu	0.22%	51.81%	19.64%	28.55%
bzip2	0.42%	32.22%	24.92%	42.86%
crafty	0.06%	33.05%	11.23%	55.72%
equake	7.30%	67.86%	10.88%	21.26%
gap	9.21%	42.55%	25.82%	31.62%
gzip	0.70%	44.17%	33.49%	22.33%
mcf	8.48%	68.78%	6.90%	24.32%
mesa	6.45%	35.76%	22.58%	41.66%
mgrid	1.37%	10.31%	19.00%	70.69%
parser	6.07%	55.25%	14.61%	30.14%
swim	2.03%	45.60%	20.25%	34.15%
twolf	0.15%	61.37%	13.67%	24.97%
vortex	1.56%	58.32%	14.73%	26.95%
vpr	0.25%	44.54%	19.27%	36.19%
wupwise	8.59%	81.75%	12.64%	5.61%

Table 3: Overhead Breakdown: Total Overhead of switching over entire execution and percentage of overhead spent flushing pipeline and switching cores

The sources of overhead can be split into flushing the pipeline of a core in order to switch execution and then copying the register file from core to core. In this table, we have marked *StallCore4* and *StallCLP+VP* as the time spent waiting to flush the pipeline during a switching decision. Finally, *Switch* is the time copying the register file over to the other core. We see that flushing the pipeline is a significant overhead. More specifically we see that for *mcf* and *equake* this overhead composes more than 80% of the total overhead. This is due to the increase in L2 misses needed to be serviced before switching. An interesting fact is that switching from the CLP+VP core back to the ILP is faster since the core is not as wide and has fewer in-flight instructions. One more source of overhead is the invalidation of shared cache lines that are a result of switching cores. We have not measured this, but we modeled it in our simulations.

6 Related Work

6.1 Memory Level Parallelism

Although we are in the era of CMP, sequential performance is important. Lots of research has focused on designing processors addressing the memory wall issue by increasing the programs MLP. Prefetching in the form of helper threads is a technique used to extract MLP [6, 32, 25, 18, 9]. Execution of the helper thread is done in-parallel on a SMT, or on a separate core for CMP architectures.

Other techniques focus on increasing the window size by unblocking the pipeline on cache misses, increasing the MLP [17, 27, 7]. In these techniques long-latency operations (and dependent instructions) are removed from the scheduling window and inserted to buffers, thus freeing resources. When the latency is resolved the instructions are reinserted into the scheduling window.

Runahead execution [19, 8], CAVA [3], and Clear [13] tolerate the long latency of L2 misses by retiring the load instruction when it reaches the head of the ROB and continuing execution despite the fact that the load has not completed. This is done by using a value prediction. When the memory request returns it re-executes the instructions or if the value was correct continue execution. Chou *et al.* [4] evaluates the effectiveness of out-of-order execution on MLP compared to in-order processors as well as the effectiveness of value predictors, branch predictors and runahead execution in enabling the extraction of more MLP. Our work contributes to how an MLP technique could be added to an AMP.

For our MLP core we implement a technique which is like CAVA along with optimizations proposed for Runahead. We picked a runahead like MLP technique because it does not require any modification to the binary. The hardware modifications also require significantly less design effort (modify cache modules) over a technique like CFP which modifies the processor pipeline. We picked a CAVA-like implementation instead of Runahead due to the benefits of Value Prediction on the load miss. This can avoid rollback and provide power savings.

6.2 Asymmetric Chip Multiprocessors

Asymmetric Chip Multi-Processor designs have been proposed as a solution to achieve higher performance per watt ratio for executing a wider range of applications [14, 16, 1, 29]. This doesn't always result to an improved performance over a homogeneous system for single threaded applications. Given the same area previous proposals [29, 1, 16, 26] achieve performance improvements when scheduling between cores at the multi-application or multi-thread level. No previous proposal has suggested that fine grain scheduling can achieve performance benefits, while only using one core at a time and running one version of the application, which is what our scheme provides.

Recent work on AMPs designed for MLP is presented by Pericas *et al.* [22]. In this work, an AMP design is composed of a fast cache core and a small in-order memory core to exploit high and low locality code respectively. By coupling the cores together an increased instruction execution window is created. Our approach is different in that the ILP and MLP cores are fully functional cores which can work independently if needed. Execution of threads is always on one core rather than spread across cores. Architectural Contesting [20] is another AMP proposal that uses a slipstream paradigm [23] to speed up sequential performance. Our approach is different since we use one thread of the application executing on one core, however both papers try to exploit phases at a fine grain level. Another slipstream paradigm using AMPs is presented in [10] where one core is of reduced complexity and the other is the correctness core. The reduced-core executes speculatively optimized code, which works as a value and branch predictor to the correctness core. This approach, however, requires recompilation of the application to create a reduced version of the program that will run on the reduced complexity core. The reduced complexity core is also not an independent functioning core. We do not need recompilation for our scheme, and both cores can function independently.

Another multi-core design for MLP is described in dual-core execution [30], where we have two homogeneous cores, coupled together with a forwarding queue to form a larger instruction execution window. The first core executes the instructions and when a long latency stall occurs, an invalid value is used to prevent the cache miss from blocking the pipeline. When instructions retire from the front processor, they are inserted into a queue and forwarded to the second processor. The front processor, besides providing the correct (due to resolving branches) instructions stream, acts as a warm up for the cache by prefetching data. In this proposal every instruction is executed twice and requires both cores on at the same time during the entire execution.

7 Conclusion

Main memory latency is still a significant performance limiting factor in today's systems. Asymmetric Multicore Processors(AMPs) offer a unique opportunity to exploit MLP by incorporating techniques onto customized cores specifically designed to exploit it. Using a detailed model of cores accurate enough to calculate detailed timing and power characteristics, we determined that narrower cores, in our case a 2-wide issue width, were more effective at exploiting Checkpointed L2 Miss Processing than a wider 4-wide issue core — providing better performance and energy efficiency across the MLP continuum.

We leveraged this finding to support Symbiotic Core Execution on an AMP. SCE is an effective scheduling mechanism because it allows MLP regions to exploit the higher performance and better power efficiency of the customized core while still leveraging the high ILP core during regions with little to no MLP. Using SCE, we achieve performance improvements of 4% and 10% over a single core MLP technique for SPECint and SPECfp with a maximum speedup of 19.5%, while at the same time reducing the energy delay^2 by 10% and 25%.

References

[1] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 29–40, New York, NY, USA, 2006. ACM.

[2] Eric Borch, Srilatha Manne, Joel Emer, and Eric Tune. Loose loops sink chips. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 299, Washington, DC, USA, 2002. IEEE Computer Society.

[3] Luis Ceze, Karin Strauss, James Tuck, Josep Torrellas, and Jose Renau. CAVA: using checkpoint-assisted value prediction to hide

l2 misses. *ACM Trans. Archit. Code Optim.*, 3(2):182–208, 2006.

[4] Yuan Chou, Brian Fahs, and Santosh Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 76, Washington, DC, USA, 2004. IEEE Computer Society.

[5] Niket K. Choudhary, Salil Wadhavkar, Tanmay Shah, Sandeep Navada, Hashem Hashemi, and Eric Rotenberg. Fabscalar. In *the Workshop on ArchitectureResearch Prototyping (WARP): held in conjunction with 36th International Symposium Computer Architecture (ISCA)*, 2009.

[6] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. Dynamic speculative precomputation. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 306–317, Washington, DC, USA, 2001. IEEE Computer Society.

[7] Adrián Cristal, Oliverio J. Santana, Mateo Valero, and José F. Martínez. Toward kilo-instruction processors. *ACM Trans. Archit. Code Optim.*, 1(4):389–417, 2004.

[8] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 68–75, New York, NY, USA, 1997. ACM.

[9] Ilya Ganusov and Martin Burtscher. Future execution: A prefetching mechanism that uses multiple cores to speed up single threads. *ACM Trans. Archit. Code Optim.*, 3(4):424–449, 2006.

[10] Alok Garg and Michael C. Huang. A performance-correctness explicitly-decoupled architecture. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 306–317, Washington, DC, USA, 2008. IEEE Computer Society.

[11] Gnu compiler collection. URL, 2009. http://gcc.gnu.org/.

[12] A. Hartstein and Thomas R. Puzak. The optimum pipeline depth for a microprocessor. In *Proceedings of the 29th annual international symposium on Computer architecture*, page 7, 2002.

[13] Nevin Kirman, Meyrem Kirman, Mainak Chaudhuri, and Jose F. Martinez. Checkpointed early load retirement. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 16–27, Washington, DC, USA, 2005. IEEE Computer Society.

[14] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 81, Washington, DC, USA, 2003. IEEE Computer Society.

[15] Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 23–32, New York, NY, USA, 2006. ACM.

[16] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 64, Washington, DC, USA, 2004. IEEE Computer Society.

[17] Alvin R. Lebeck, Jinson Koppanalil, Tong Li, Jaidev Patwardhan, and Eric Rotenberg. A large, fast instruction window for tolerating cache misses. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 59–70, Washington, DC, USA, 2002. IEEE Computer Society.

[18] Andreas Moshovos, Dionisios N. Pnevmatikatos, and Amirali Baniasadi. Slice-processors: an implementation of operation-based prediction. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 321–334, New York, NY, USA, 2001. ACM.

[19] Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Efficient runahead execution: Power-efficient memory latency tolerance. *IEEE Micro*, 26(1):10–20, 2006.

[20] Hashem H. Najaf-abadi and Eric Rotenberg. Architectural contesting: exposing and exploiting temperamental behavior. *SIGARCH Comput. Archit. News*, 35(3):28–35, 2007.

[21] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 206–218, New York, NY, USA, 1997. ACM.

[22] Miquel Pericas, Adrian Cristal, Francisco J. Cazorla, Ruben Gonzalez, Daniel A. Jimenez, and Mateo Valero. A flexible heterogeneous multi-core architecture. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[23] Zach Purser, Karthik Sundaramoorthy, and Eric Rotenberg. A study of slipstream processors. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 269–280, New York, NY, USA, 2000. ACM.

[24] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC Simulator, January 2005. http://sesc.sourceforge.net.

[25] Amir Roth. *Pre-execution via speculative data-driven multithreading*. PhD thesis, 2001. Supervisor-Sohi,, Gurindar S.

[26] Juan Carlos Saez, Manuel Prieto, Alexandra Fedorova, and Sergey Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 139–152, New York, NY, USA, 2010. ACM.

[27] Srikanth T. Srinivasan, Ravi Rajwar, Haitham Akkary, Amit Gandhi, and Mike Upton. Continual flow pipelines. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 107–119, New York, NY, USA, 2004. ACM.

[28] J.E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W.R. Davis, P.D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal. Freepdk: An open-source variation-aware design kit, 2007.

[29] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 253–264, New York, NY, USA, 2009. ACM.

[30] Huiyang Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 231–242, Washington, DC, USA, 2005. IEEE Computer Society.

[31] Huiyang Zhou and Thomas M. Conte. Enhancing memory level parallelism via recovery-free value prediction. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 326–335, New York, NY, USA, 2003. ACM.

[32] Craig Zilles and Gurindar Sohi. Execution-based prediction using speculative slices. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 2–13, New York, NY, USA, 2001. ACM.