Efficiently Exploiting Memory Level Parallelism on Asymmetric Coupled Cores in the Dark Silicon Era

GEORGE PATSILARAS, NIKET K. CHOUDHARY, and JAMES TUCK,
North Carolina State University

Extracting high memory-level parallelism (MLP) is essential for speeding up single-threaded applications which are memory bound. At the same time, the projected amount of dark silicon (the fraction of the chip powered off) on a chip is growing. Hence, Asymmetric Multicore Processors (AMP) offer a unique opportunity to integrate many types of cores, each powered at different times, in order to optimize for different regions of execution. In this work, we quantify the potential for exploiting core customization to speedup programs during regions of high MLP. Based on a careful design space exploration, we discover that an AMP that includes a narrow and fast specialized core has the potential to efficiently exploit MLP.

Using the results of our analysis, we design an AMP with both an MLP and ILP specialized core, and we propose a hardware-level, application steering mechanism called Symbiotic Core Execution (SCE). SCE detects MLP phases by monitoring the L2 miss rate of the application, and it uses that information to steer the application to the best core. Interestingly, we show that L2 miss rates are important for deciding when an MLP region begins and when it ends. As a program runs, its execution migrates to a core customized for MLP during regions of high MLP; when the region ends, it is re-scheduled on the core that fits the application characteristics.

Compared to a monolithic core optimized for both modes of operation, our AMP design provides a harmonic mean performance improvement of 5.3% and 6.6% for SPEC2000 and SPEC2006, respectively, with a maximum speedup of 14.5%. For the same study, it achieves a 18.3% and 21.1% energy delay² reduction for SPEC2000 and SPEC2006, respectively. Our findings yield an important message for designing AMPs with specialized cores: core customization enables efficient exploitation of MLP, and application steering mechanisms for MLP are simple to implement and effective.

Categories and Subject Descriptors: C.0 [General]; C.1.0 [Processor Architectures]: General

General Terms: Design, Performance

Additional Key Words and Phrases: Memory level parallelism, multicore, asymmetric multicore processor, dark silicon

ACM Reference Format:

Patsilaras, G., Choudhary, N. K., and Tuck, J. 2012. Efficiently exploiting memory level parallelism on asymmetric coupled cores in the dark silicon era ACM Trans. Architect. Code Optim. 8, 4, Article 28 (January 2012), 21 pages.

DOI = 10.1145/2086696.2086707 <http://doi.acm.org/10.1145/2086696.2086707>

1. INTRODUCTION

Chip Multicore Processors (CMPs) have superseded large monolithic processors due to their excessive design complexity and high power consumption. As a result, most CMP designs have focused on increasing the number of cores and re-organizing the memory hierarchy. However, due to power limitations, in the near future a large portion of the

Author's address: G. Patsilaras, N. K. Choudhary and J. Tuck, North Carolina State University; email: {gpatsil, nkchoudh, jtuck}@ncsu.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2012 ACM 1544-3566/2012/01-ART28 \$10.00

DOI 10.1145/2086696.2086707 <http://doi.acm.org/10.1145/2086696.2086707>

chip will have to be powered off [Esmaeilzadeh et al. 2011] at any given time. This means that architects should not envision designs in which every transistor remains turned on permanently, but rather ones in which portions of the chip are judiciously turned on/off depending on the characteristics of a workload. For example, a specialized processing core could be off most of the time and only enabled when it is really needed. Hence, there is growing interest in Asymmetric Multicore Processors (AMPs). AMPs are typically composed of single-ISA, heterogeneous cores with varied microarchitectural features. AMPs offer a unique opportunity to speedup single threads by specializing cores to distinct phases of many different applications. This comes with some caveats which are 1) how do we know which core out of a selection is the highest performing core and 2) when do we switch to this preferred core.

Memory level parallelism (MLP) has been proposed as a way to boost the performance of applications that stall frequently due to misses in the last level cache (LLC). Rather than waiting for one access at a time, the goal of MLP techniques is to exploit available memory bandwidth to request many memory accesses at once. A variety of hardware-only MLP enhancing techniques have been proposed [Collins et al. 2001; Moshovos et al. 2001; Ganusov and Burtcher 2006; Mutlu et al. 2003; Dundas and Mudge 1997; Ceze et al. 2006; Kirman et al. 2005; Zhou and Conte 2003; Lebeck et al. 2002; Srinivasan et al. 2004; Zhou 2005]. These techniques are advantageous since they can transparently accelerate sequential codes but they are limited by their high energy consumption. Some of these techniques leverage precomputation to issue loads in advance of when they are needed. Other MLP techniques leverage hardware within a single core to detect a long latency load; then speculate past the load to generate overlapping misses [Ganusov and Burtcher 2006; Ceze et al. 2006; Kirman et al. 2005; Zhou and Conte 2003]. Multithreaded approaches have been considered which automatically construct prefetching threads [Ganusov and Burtcher 2006], or tightly couple two cores to act as a large instruction window [Zhou 2005]. Given the importance of tolerating long latencies to memory *within a single thread*, future processors will likely incorporate techniques to overlap long latency misses.

AMPs are an interesting design space in which to consider the addition of such MLP techniques. Since many applications have ILP and MLP phases during their execution, cores on an AMP could be leveraged to exploit the behavioral differences among these phases. Furthermore, the power and design complexity of new architectural features to exploit MLP can be mitigated by designing a custom core for MLP.

In this paper, we study the behavioral characteristics of high MLP code and determine that it can benefit from core customization. This leads us to design a new system called Symbiotic Core Execution (SCE), in which at least one core is customized for the characteristics of code with high ILP and another core is customized for the characteristics of higher MLP. With the help of transparent and judicious scheduling mechanisms in hardware, we can exploit the characteristics of the MLP-customized core during regions of high MLP; when ILP dominates, a *preferred* core is used. This paper makes the following contributions.

- We are the first to characterize the *performance* benefits of core customization on an AMP for MLP regions. Specifically, we use an analysis of L2 miss rates to compare program regions with similar behavior on different core designs. Contrary to conventional wisdom, even with frequent L2 misses, we show that frequency and core width combined are an important design characteristic. In this space, we show definitive performance benefits of a narrow core over a wider core in sequential program regions.
- We propose Symbiotic Core Execution (SCE), a technique which benefits from new opportunities created by an increasing fraction of dark silicon on future chips. By

```

...
while( arcin ){
    tail = arcin->tail;
    if( tail->time + arcin->org_cost > latest )
    {
        arcin = (arc_t *)tail->mark;
        continue;
    }
    red_cost = arc_cost - tail->potential + head_potential;
    if( red_cost < 0 ){
        ...
    }
    arcin = (arc_t *)tail->mark;
}

```

Fig. 1. Function price_out.impl(network.t net) in mcf.

using an AMP with one core customized for ILP and one specialized for regions of high MLP, we are able to exploit the fine-grained phase behavior of applications while powering off the unused core.

- For SCE, we identify a hierarchical, hardware-level scheduling mechanism which judiciously switches cores to exploit regions of high MLP on the customized MLP core without incurring a high switching overhead. We also detect when the MLP region ends in order to switch back to the preferred core at an appropriate time. We show that it is sufficient to look at L2 miss rates in order to identify MLP regions.
- Finally, our approach provides an average improvement of 5.3% and 6.6% for SPEC2000 and SPEC2006 applications, respectively, with a max speedup of 14.5%, compared to implementing the MLP technique on a single core. Our findings yield an important message for designing AMPs with specialized cores: core customization enables efficient exploitation of MLP, and application steering mechanisms for MLP are simple to implement and effective.

The rest of the paper is organized as follows. Section 2 gives some background on MLP techniques and AMP benefits; Section 3 presents a detailed study of different core designs for varying levels of MLP. Section 4 describes our Symbiotic Core Execution architecture, 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. Application Limitations for High Performance

Last level cache misses (called L2 misses here on) are an important factor affecting performance of applications. Pointer-chasing, large working sets, or frequent changes in phases can result in high L2 misses. Figure 1 shows a code fragment from mcf, an application with lots of L2 misses. This code represents a pointer-chasing loop dominated by long-latency memory operations. Typically, a large instruction window can help L2 misses, however, in this code, the instructions following the load depend upon it. Even though an out-of-order processor can continue executing instructions while the miss is handled, the lack of independent instructions prevent it from doing so.

2.2. Effects of a Core's Width on Cycle Time

A large instruction window along with a complex microarchitecture, might enhance IPC, but at the same time it could increase the clock period. For instance, increasing

Fig. 2. Scaling of Issue Queue in terms of delay.

the size of the issue window and issue width can boost IPC for applications with abundant ILP. However, that may not translate into higher performance because the clock period may increase to accommodate the larger content addressable memory and deeper select tree.

Fabscalar [Choudhary et al. 2011], is a state of the art tool that enables architects to synthesize customized designs and evaluate the effects of different designs in terms of frequency, area and power. Using Fabscalar, we can synthesize a Verilog model of an arbitrary superscalar processor and analyze how the size of key processor structures can affect frequency. Figure 2 shows the impact of increasing the issue window on the delay 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 on different key structures of processors (ROB size, LSQ Unit, Core Width etc). Our search focused on identifying cores with the highest frequency with issue widths of one, two, and four. 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 1.66GHz, the 2-wide core frequency was 2.22GHz and the 1-wide core had a maximum frequency of 2.5GHz (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. [Palacharla et al. 1997], in order 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 for 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. In the rest of the article, we use the core designs found by Fabscalar.

2.3. Checkpointed L2 Miss Processing with Value Prediction: CLP+VP

Another way to virtually increase the instruction window is by using lookahead-like paradigms, such as Runahead Execution [Mutlu et al. 2006], CAVA [Ceze et al. 2006], which try to speculatively extract high amounts of MLP. Our basic mechanism, shown in Figure 3(a) adopts some features from CAVA [Ceze et al. 2006] and Clear [Kirman et al. 2005]. 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

Fig. 3. MLP regions.

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 exits the speculative epoch if the prediction was correct. 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 [Ceze et al. 2006]. We will refer to this basic strategy as Checkpointed L2 Miss Processing with Value Prediction (*CLP+VP*) show. *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 3(b) shows the *CLP+VP* architecture with the required Speculative Data Cache [Ceze et al. 2006]. 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.

2.4. Core Customization

Core customization is an attractive way of utilizing the real estate of CMPs due to application diversity. Previous research has focused on exploiting applications diversity given different cores [Kumar et al. 2004]; however, there have been few studies on core preference for a given architectural technique. Looking at problems such as pointer-chasing and solutions like *CLP+VP* a question someone might have is what impact this has on core selection for an AMP. There are many factors that make this analysis non-trivial and difficult to anticipate.

- The trade-off between core-width and frequency is largely driven by ILP. While MLP and ILP are not strictly correlated, conventional wisdom argues that MLP tends to come in regions of lower ILP. Hence, this suggests narrower cores may be better.
- Latencies to memory are fixed regardless of frequency. Therefore, the frequency of the core may not matter since the execution time of MLP regions may be ultimately dominated by latency to memory. Hence, the fixed latency to memory suggests that an average sized core – in other words, nothing too fast, and nothing too slow – will work just as well as anything else.
- Finally, the accuracy of techniques like *CLP+VP* to speculate over L2 misses using predicted values will determine how much MLP is exploited. If misspeculations are infrequent, *CLP+VP* behaves like a very large window, which could favor wider cores. However, if misspeculations are frequent or if regions executed speculatively tend to have lower ILP, narrow cores could be favored.

Table I. Asymmetric Core Configurations

Label	Configuration
<i>Core4</i>	4-wide 1.66GHz Core
<i>Core2</i>	2-wide 2.22GHz Core
<i>Core1</i>	1-wide 2.5GHz Core
<i>Core2 CLP+VP</i>	2-wide 2.22GHz Core with CLP+VP
<i>Core4 CLP+VP</i>	4-wide 1.66GHz Core with CLP+VP
<i>SCE CLP+VP</i>	4-wide 1.66GHz Core+2-wide 2.22GHz CLP+VP core

Fig. 4. Virtual instruction window of CLP+VP.

Through detailed analysis in the next section, we hope to better understand these factors.

3. MOTIVATION

3.1. Effects of MLP Techniques on Different Cores

One of the benefits of *CLP+VP* is that it achieves a larger virtual instruction window by speculating over a L2 miss. Figure 4 shows the effect that this will have for a 4-wide core and a 2-wide core. The dark solid line cutting the instruction window shows the actual instruction window and how the virtual instruction window will be larger. Regardless of the width both cores will achieve a large virtual instruction window.

When comparing a 4-wide and a 2-wide core, we can assume that if there are no branches or dependent instructions the 4-wide core would achieve a larger virtual instruction window. On the other hand, due to instructions' dependencies, the virtual instruction window of the 2-wide core can end up being larger due to its faster cycle time. As a result, a 2-wide core, besides issuing L2 misses earlier, can also issue more L2 instructions in the same time of execution. In the next section, we perform a study to determine which behavior is more prevalent for regions of high and low MLP.

Another benefit for a faster clock frequency is when *CLP+VP* mispredicts a value. In that case, the processor needs to rollback to the checkpointed state and re-execute the instructions. This is done quicker on a 2-wide core giving a faster recovery time.

3.2. Exploring MLP on Different Core Widths

In this section, we investigate the behavior of different core designs, shown in Table I, in code regions sensitive to MLP techniques. Our goal is to establish a relationship between core performance and suitability for exploiting MLP on the same code 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 categorize these regions according to their L2 miss rate. Hence, for a wide range of applications, we can average out behavior based solely on the potential for MLP.

Figure 5 plots the performance of the *Core2*, *Core1*, *Core4+CLP+VP*, *Core2+CLP+VP*, and *Core1+CLP+VP* designs normalized to the *Core4* design. The dashed lines show all the cores without CLP+VP, while solid lines are taking CLP+VP into account. Each bin on the x-axis indicates a different L2 miss rate (L2 misses/10K instructions). Each point on the y-axis is the speedup over *Core4* for all code regions with a specific L2 miss

Fig. 5. Performance of different core widths normalized to Core4.

Fig. 6. Performance versus MLP potential.

rate. Since these cores have no additional MLP technique, they are only exploiting the MLP available from out-of-order execution.

In Figure 5 we can identify three regions. The first region, identified as *ILP + low MLP*, has no to very few misses. Most high ILP applications spend the majority of their execution in this region (98% for ammp). In this region 4-wide core designs have a clear advantage due the high ILP exploited by the width of the cores. The second region is identified as *Medium ILP & MLP*. In this region we see that ILP is diminished somewhat by the increasing number of L2 misses. Applications executing on the 4-wide cores (*Core4* and *Core4+CLP+VP*) are faster than *Core1*, and *Core1+CLP+VP*, however we see that designs *Core2*, and *Core2+CLP+VP* are competitive. When comparing the 2-wide core designs to the 4-wide cores, we can consider the 2-wide cores a better design due to the power savings of a smaller core (for an explanation on how higher frequency can save on power see Section 5).

The third region identified as *MLP+ Low ILP* is the most interesting region. We see that the 2-wide core designs gain a slight advantage. In Figure 6(b) we show the actual IPC and instructions per second (IPS) for the 100 L2 miss bin. These regions have high amounts of L2 misses per sampled region, however what is interesting is that here we see clearly the core width versus core frequency trade-off.

For the third region, L2 miss rates are high. One might expect L2 misses to dominate the execution time (regular instructions would complete while misses are being serviced), and since the time to access main memory is constant, the cores should perform the same. However, this is not the case. Figure 6(a) shows on the x-axis the distance between issued instructions which miss in the L2. On the y-axis we have the total percentage of load instructions which have an issue distance of 'x' or less from a previous L2 miss. The three lines represent different L2 miss rate bins from Figure 5. From

Fig. 7. Architecture of Tile and AMP.

this graph, it is clear that regardless of the amount of L2 misses in every 10K bin, L2 misses cluster close together. This clustering is important for two reasons. First, it explains why MLP techniques can benefit sequential performance. Second, clustering shows us that execution time for these bins is actually composed of memory access regions and non-memory access regions. The conventional wisdom fails to account for application and core-level effects in regions of high L2 miss rates, however, our analysis does account for them. Therefore, we see that the *MLP+Low ILP* region can exploit asymmetry.

4. SYMBIOTIC CORE EXECUTION

The goal of this paper is to design an AMP which can identify MLP regions on the fly and schedule execution to the best available core. To do this, we propose 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 necessary, they can be used together for a greater performance advantage.

SCE is based on AMP architectures and leverages a loose coupling of two asymmetric cores. In our design, one core will provide higher ILP, and the other core will be customized for MLP. Sequential performance is improved by assigning fine-grained phases of an application to the appropriate core. A key challenge of SCE is efficient fine-grained scheduling. In the rest of this section, we provide an overview of the SCE architecture and a description of our scheduling algorithm.

4.1. Overall Architecture

SCE's mechanism is implemented on a homogeneous ISA, heterogeneous-tile CMP. Figure 7(a) shows an example SCE tile and Figure 7(b) shows the homogeneous-tile CMP implementation. Due to power constraints and the large areas of a chip that will have to remain powered off in the near future [Esmailzadeh et al. 2011], we can power on only one core of the tile at a given time. We refer to this as SCE mode. In SCE mode, the OS sees one logical core per tile, and hardware can initiate fast context switching among the symbiotic cores. In the case where the number of threads is more than the number of cores per tiles, we assume that SCE is disabled, and the chip can enable different combinations of types of cores to fit applications needs for a given power budget. Scheduling applications in this case would be similar to previous work such as [Kumar et al. 2004].

Since SCE mode (one core per tile powered at a time) is our new support, we will primarily focus on how it works and do not discuss the different combinations of types of cores that could fit to a given power budget. In SCE mode, a scheduling algorithm designed to exploit the MLP core will determine when a core switch is needed. At that time, it stalls the fetch engine in preparation of collecting the program's context. Once the L2 miss reaches the head of the reorder buffer, the processor squashes the remaining instructions, and performs the context switch between the cores by copying the register file from one core to the next. We design the context switch to begin as soon as the head of the ROB stalls by a long latency instruction. That way, the context switch delay is overlapped by the servicing of an L2 miss. Once the context switch is finished, execution is resumed. If we switched because of a long latency instruction at the head of the re-order buffer, then that instruction is re-executed.

Our hardware level context switch happens only during SCE mode. Furthermore, the idle core will always be available, so our context switch mechanism can be fast. Even though we have hardware support for switching, many overheads may be incurred on a switch, such as cache warm-up penalty, cache invalidations, effects on the TLB, and poor branch predictor history. These effects are one reason why building an efficient SCE scheduler is important.

4.2. SCE Mode Scheduling

Our goal is for SCE mode scheduling to work well for a wide range of applications but especially for those that contain significant amounts of MLP. This is challenging because there are hierarchical, competing concerns. At the Core-level, we do not know a priori which node will usually provide the best performance. We must determine a *preferred* core on the fly. Furthermore, this preferred core can change throughout execution, therefore we must adapt to the coarse-grained phase of the running application. In addition, at the fine-granularity of MLP phases, we will switch to the MLP core to exploit its MLP technique, and we must switch back to the preferred core in regions of low ILP. Failure to identify a good start or end to the MLP region could take the application away from its preferred core for too long. Finally, even within MLP epochs, the MLP technique should be used only when it can actually provide a performance benefit. A good scheduling algorithm will manage all of these concerns.

In the rest of this section, we describe our hierarchical scheduling strategy in three parts: *Preferred Core scheduling*, *MLP Region Detection*, and *MLP Epoch Scheduling*.

4.3. Preferred Core Scheduling

Figure 8 shows pseudocode for our overall core scheduling strategy. Because we wish to choose the best core even when not in an MLP region, much of the code is concerned with picking the *preferred* core. Overall, the algorithm is divided into training mode and non-training mode. Every 50 million cycles, our algorithm enters training mode and picks a new preferred core. We selected 50 million cycles in order to have a large enough period for our preferred core scheduling policy to be not so frequent. We could just enter this scheduling mode if we had a way to detect phase changes such as described in Sherwood et al. [2003]. The first if-condition controls the actions during training mode. For each core in the AMP tile, we run for a minimum of 100K cycles and record the instructions per second (IPS) in a hardware register. Note, we must use IPS to account for different operating frequencies. Once all cores in the tile have been trained, we select the core with the highest IPS. Note that we do not disable the MLP technique when training on the MLP core. While this may distort the IPS compared to running without the MLP technique, it usually does not alter the preferred cored decision.

```

while(1) {
    if (in training period) {
        if (execution was on both cores) {
            set preferred core based on IPS
            exit training
        } else {
            force training period of 100K cycles on other core
        }
    } else {
        ----- MLP region Scheduling begin-----
        if (MLP region) {
            -----MLP Epoch scheduling begin-----
            schedule on CLP+VP core
            -----MLP Epoch scheduling end -----
        } else {
            -----Preferred Core scheduling begin-----
            schedule on preferred core
            -----Preferred Core scheduling end-----
        }
        -----MLP region Scheduling end-----
    }
    if (50 million cycles pass)
        enter training mode
}

```

Fig. 8. SCE scheduling algorithm.

When not in training mode, non-MLP regions will run on the preferred core by default until an MLP region is encountered. When we enter an MLP region, execution switches to the MLP core. As long as the MLP region is active, execution remains there. However, once the MLP region ends, execution returns again to the preferred core.

Overall, we expect this scheduling algorithm to work well for a wide range of programs. Since it retrains the preferred core on the AMP every 50 million cycles of execution, it is likely to adapt to changing program behavior whether or not MLP regions occur. Furthermore, if an application always benefits from being on the MLP core, this algorithm will ensure that it is never scheduled off that core except for short regions of training. However, it critically relies on judicious MLP Region detection. If MLP regions are triggered too frequently, it can add significant overhead and hurt performance. In the next section, we consider the design of the MLP Region Detection policy.

4.4. MLP Region Detection

4.4.1. Responsive Switching to MLP Core. The first goal of our region 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 crossover point at 10 L2 misses per 10K instructions. So, as a heuristic, we assume that we need to observe misses at that rate in order 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 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 9(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 leading to significantly more overhead and loss of performance. However, if N_{inst} is too large, it increases the likelihood that we wait too long to switch and miss

Fig. 9. SCE scheduling illustration.

good scheduling opportunities. After our experiments from Section 3 we found that in regions of 90 misses per 10K instructions our MLP core is better so we set N_{inst} equal to the time it would take for our system to execute approximately enough instructions to observe this miss rate. After a series of experiments we concluded that $N_{inst} = 3000$ works well (approximately 9 misses $\times$ memory latency + the cycles to execute 1000 instructions at a CPI of 1). The hardware cost associated with the counters to measure L2 misses is very small and already present in current microprocessors.

4.4.2. Lazy Switching Back to the Preferred Core. Once running on the MLP core, our scheduler evaluates a) if it was a correct decision to switch and b) if there is no longer a benefit from MLP. It is important to remain on *Core2+CLP+VP* long enough to exploit any MLP. However, it is desirable to correct erroneous decisions 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. 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 preferred 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 9(b) illustrates this policy. We found that $N_{inst} = 3000$, $N_{stay} = 700$ and $N_{ext} = 3000$ work well. Note the rate of L2 misses required to switch back to the preferred core is lower than that required to switch away. Our scheduling policy is more sensitive to the N_{inst} and r_{miss} parameters.

4.5. MLP Epoch Scheduling

Previous work has shown that MLP techniques need to be smart on how speculation is performed on L2 misses [Mutlu et al. 2003; Ceze et al. 2006]. In order for *CLP+VP* to be effective, the value prediction must be accurate in order to avoid polluting the cache and fetching instructions down the wrong path. We use a last-value predictor indexed by a hash of the PC and the branch history and we use a confidence estimator with a two-bit saturation counter indexed by the PC, just like in Ceze et al. [2006]. If the confidence on a prediction is too low, we do not speculate. Furthermore, we geometrically accumulate the confidence estimate for all the speculative load values created during an epoch. Once the accumulated confidence drops below a threshold, we stop speculating.

Another source of inefficiency is speculating over the shadow of an L2 miss that overlaps with no other L2 misses. If running on the ILP core (when it is the preferred core), this inefficiency is prevented because we are unlikely to switch to the MLP core for isolated L2 misses. When we are on the the MLP core, we avoid speculation because our value prediction confidence estimator will eliminate solitary misses with low prediction accuracy.

5. EVALUATION OF SCE

5.1. Experimental Setup

To evaluate our AMP proposal we used SESC [Renau et al. 2005], a detailed, microarchitectural simulator, and compiled SPEC2000 and SPEC2006 applications using a MIPS cross compiler built from GCC 4.4 [gcc 2009] with -O3. Our simulations use the ref inputs set. We skip the initialization stage of applications and simulate at least 600 million correct instructions for SPEC2000 and 300 million for SPEC2006. We used as many of the SPEC2000 and SPEC2006 as we could to guarantee a mix of applications with both low-MLP and high-MLP behavior; also, some were excluded due to limitations of the cross-compiler or simulator.

We also used the FabScalar framework [Choudhary et al. 2011] to weigh the cost of different superscalar designs in terms of clock period, area, and power (static and dynamic). The FabScalar framework synthesizes 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 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 Cadence SoC Encounter V7.1, using the FreePDK OpenAccess 45nm Standard Cell Library [Stine et al. 2007] to synthesize our different designs in order to estimate timing, power and area. SESC supports power modeling and we feed the power characteristics of each structure with data from Fabscalar. To keep the design space within limits, we do not modify the voltage supply, which could affect frequency, and restricted the total pipeline depth to 14 (from Fetch to Commit) for all compositions of the processors used in the evaluation. We believe that a 14-deep pipeline is appropriate, since it can also be found in contemporary high performance microprocessors [Wechsler 2006].

5.2. Power Modeling

Since we allow different core designs to operate at different frequencies, it is important to carefully consider trade-offs in terms of power and performance in our evaluation. In this section, we describe the methodology for calculating the operating frequencies of each core in all simulated configurations.

The processors evaluated in this study have different peak power characteristics. To see why, we will briefly discuss the overall power model for a processor. Power consumption of a processor consists of the static and dynamic component ($P_{\text{Total}} = P_{\text{Dynamic}} + P_{\text{Static}}$). Dynamic power consumption on a chip is given by Eq. (1), where P is the power, C is the effective load, α is the switching activity factor, V is the input voltage and, F is the processor frequency.

$$P_{\text{Dynamic}} = \alpha CV^2 F \quad (1)$$

The peak power is calculated much like P_{Total} except that an average switching factor is replaced with the worst case dynamic load for the chip. When running on an SCE configuration, the peak power is:

$$P_{\text{PeakSCE}} = \text{Max}(P_{\text{DynPeakCore2}}, P_{\text{DynPeakCore4}}) + P_{\text{StaticCore2}} + P_{\text{StaticCore4}} + P_{\text{PeakAllElse}} \quad (2)$$

Since it is the case that $P_{\text{DynPeakCore2}} < P_{\text{DynPeakCore4}}$, the equation can be simplified to only consider the peak dynamic power of Core4. Also, $P_{\text{PeakAllElse}}$ is the remaining

Table II. Core Details. Cycle Counts are in Processor Cycles

Core Setup Origin Frequency	Multicore Fabscalar 1.66 GHz	Single Core Scaled Peak 1.68 GHz	Multicore Fabscalar 2.22 GHz	Single Core Scaled Peak 2.31 GHz	Multicore Scaled Peak 2.30 GHz	Single Core Fabscalar 2.5 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	
Main Memory RT	184 cycles	186 cycles	246 cycles	256 cycles	255 cycles	277 cycles
Core Switch Delay	37 cycles		50 cycles			55 cycles
Area(mm ²)	1.636		1.118			0.718
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						
Core4 Cache: IL1 and DL1: Size=32KB; Assoc=4-way; Line size=64B; RT=3 cycles						
Core2 Cache: IL1 and DL1: Size=16KB; Assoc=4-way; Line size=64B; RT=3 cycles						
Shared L2 Cache: Size=2MB; Assoc=8-way; Line size=64B; RT=10 cycles						
BTB: size=2K; Assoc=2-way, Supply Voltage= 1.1V, Tech=45nm						
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_{ext} =3000cycles						
CLP+VP configuration: OPB = 128 entries Max. Outs. preds./Instr.=512/3072						
BHLV+GLV predictor, table size=4096 entries each						

dynamic and static power from interconnect and cache. Compared to a single Core4 design, we have added extra leakage power.

In order to make a fair comparison, we set the peak power of the chip to that of the SCE configuration. Since the single core configurations cannot reach this peak power, we use the Fabscalar tool to identify a scaled voltage and frequency that boosts their power consumption to match the peak power of the SCE configuration. Interestingly, we use this same technique to provide our own design an advantage. When running on Core2, the peak power is over-estimated by the difference in $P_{DynPeakCore2}$ and $P_{DynPeakCore4}$. Hence, SCE can also exploit scaled voltage and frequency when running on the smaller core.

We have performed a careful analysis of the peak power for each evaluated processor using the Fabscalar framework. Figure 10 shows the breakdown for each component out of $P_{PeakSCE}$. Note, since we are comparing peak power not average power, the fraction of leakage power is relatively small. In Figure 10(a), we can see the power due to Core2 (Leakage Core2+L1, Dynamic 16K L1) contributes only about 1.66% to the peak power. This amount shows up in Figure 10(c) as Unused Peak Power when evaluating a single Core4. Hence, Core4 can scale its voltage and frequency to consume 1.66% more dynamic power. As shown in Figure 10(b), the difference in peak power when executing on Core4 versus Core2 on SCE is 13.01%; this is primarily due to the difference between the dynamic power for a 4-wide core compared to a 2-wide core. Hence, when executing on Core2 during SCE, the voltage and frequency can be scaled to consume 13.01% more dynamic power.

Table II shows the resulting frequencies in our experiments. The *Fabscalar* columns show the frequency reported by the Fabscalar tool. This is used for Core4 in SCE mode. The *Single Core Scaled Peak* column shows the adjustment to frequency when running on a single core (no SCE). Finally, the *Multicore Scaled Peak* column shows the adjustment to frequency when execution is on Core2 in SCE mode.

Table II shows the SESC configuration parameters used in our experiments. Labels used in the graphs are explained in Table III along with the configurations executed.

Table III. Core Configurations

Label	Symmetry	Configuration
<i>Core4</i>	Symmetric	4-wide 1.68GHz core (Base)
<i>Core2</i>	Symmetric	2-wide 2.31GHz core
<i>Core2+CLP+VP</i>	Symmetric	2-wide 2.31GHz CLP+VP core
<i>Core4+CLP+VP</i>	Symmetric	4-wide 1.68GHz CLP+VP core
<i>SCE CLP+VP</i>	Asymmetric	4-wide 1.66GHz core+2-wide 2.30GHz CLP+VP core
<i>SCE none</i>	Asymmetric	4-wide 1.66GHz core+2-wide 2.30GHz core (Base AMP)
<i>SCE All</i>	Asymmetric	4-wide 1.66GHz CLP+VP core +2-wide 2.30GHz CLP+VP core
<i>SCE4</i>	Asymmetric	4-wide 1.66GHz Core+4-wide 1.66GHz CLP+VP core
<i>SCE imm</i>	Asymmetric	4-wide 1.66GHz Core+2-wide 2.30GHz CLP+VP core with imm. switch

Fig. 10. Power breakdown of each component in a multicore.

5.3. Performance Compared to Symmetric Multicore Processors

Figure 11 shows the speedups of SCE over *Core4*, our base case. We see that SCE with *CLP+VP* delivers an overall speedup of 1.14 for SPEC2000 applications and 1.10 for SPEC2006 over *Core4*. Looking at individual benchmarks we see that 19 out of 25 benefit from SCE. For benchmarks with no benefits, we see that they are characterized by low L2 miss rates. Finally, *applu*, *mesa*, *mgrid*, *vortex*, *gromacs*, and *dealIII*, have a slight degradation due to regions of low MLP and low value prediction accuracy during the MLP regions.

When compared to a single core with the CLP+VP technique, we see that our design is better. *SCE CLP+VP* is better than *Core4 +CLPVP* by 5.3% and 6.6% for SPEC2000 and SPEC2006 applications, respectively, with a maximum of 14.5%. The benefits come from the faster 2-wide CLP+VP core achieving better performance than the 4-wide CLP+VP for high MLP regions and the use of *Core4* to complement 2-wide CLP+VP core's for regions of low to no MLP.

Although not included here for brevity, we also performed an experiment in which the 2-wide core in *SCE CLP+VP* does not use the *Scaled Peak* frequency, rather it uses the Fabsclar reported frequency. In this configuration, *SCE CLP+VP* performs marginally worse than the *Scaled Peak SCE CLP+VP* configuration (roughly 1.5% worse on SPEC2000 and 1% on SPEC2006), but the overall performance trend remains.

5.4. Power and Energy Delay²

Figure 12(a) shows the power consumption for each design we evaluated. This includes the static and dynamic power for the occupied core, and cache hierarchy as well as the static power for the core not used. Caches remain active for the entire execution due to cache coherence, and so we modeled the static and dynamic power during the entire execution for regular accesses and invalidations triggered by the coherence protocol. The graphs show, our *SCE CLP+VP* design consumes less power compared to

Fig. 11. Speedup for SCE CLP+VP, Core2, Core2 CLP+VP, and Core4 CLP+VP normalized to Core4.

(a) Power for SCE CLP+VP, Core2, Core2 CLP+VP, and Core4 CLP+VP normalized to Core4

(b) Energy Delay² for SCE CLP+VP, Core2, Core2 CLP+VP, and Core4 CLP+VP normalized to Core4

Fig. 12. Power and Energy Delay² normalized to Core4.

Core4+CLP+VP by 18.9% and 15.6% for SPEC2000 and SPEC2006, respectively. SCE however, does require more power than *Core2+CLP+VP*, but this is a direct consequence of using *Core4* to accelerate some non MLP regions. For benchmarks with high MLP regions such as *mcf* and *equake*, we see that average power is higher on the *Core2+CLP+VP* than *Core4* due to speculation and re-executed instructions.

Figure 12(b) displays the energy delay² for each configuration. Overall, *SCE CLP+VP* reduces the energy delay² by 20.1% and 23.1% for SPEC2000 and SPEC2006 benchmarks over *Core4*. When compared to the *Core4+CLP+VP* design, the energy delay² savings are 18.3% and 21.1% for SPEC2000 and SPEC2006, respectively. Note that *equake*, *mcf*, *swim* and *gcc* are particular advantageous when considering energy delay² because their high performance offsets their high power consumption. An interesting observation here is that on average 56% of the average power is from dynamic power while the remaining 44% is due to static power.

Fig. 13. Speedup for SCE all, SCE none, SCE4, and SCE imm normalized to *SCE none*.

5.5. Performance Compared to Asymmetric Multicore Processors

The benefits of SCE also exist when compared to other AMP configurations which vary in terms of frequency or MLP techniques. Table III shows the configurations we compare against. *SCE All*, has the *CLP+VP* added to the 4-wide core as well as the 2-wide core. In this design, *CLP+VP* may be used at any time. *SCE none* has a 2-wide 2.30GHz frequency and a 4-wide 1.66GHz frequency neither of which have *CLP+VP*; the 2-wide is used for MLP regions. *SCE4* has a 4-wide base core along with a 4-wide *CLP+VP* core for the MLP regions. Finally, *SCE imm* is our SCE core configuration with a scheduler which switches to the MLP core at the first L2 miss (as opposed to using L2 miss rate like *SCE CLP+VP* does).

Figure 13 shows the speedup of each design compared to the Base AMP design, shown as *SCE none*. Overall, our design is 4.7% and 3.3% better for SPEC2000 and SPEC2006 when compared to the base AMP with no MLP technique implemented. Our SCE design is also better than *SCE4* by 4.8% and 8% for SPEC2000 and SPEC2006. When comparing against *SCE imm* we see that *SCE CLP+VP*'s performance is better by 6.3% and 9.9% over SPEC2000 and SPEC2006. This is an important indicator to why our scheduling mechanism is effective. It shows how the miss rate is more accurate for detecting MLP regions, than making scheduling decisions solely based on a single L2 miss.

When compared to *SCE All* we achieve identical performance, which indicates that our scheduler hardly misses any MLP regions. An added benefit of our scheme is the reduction in the instructions added due to value misprediction and using MLP techniques on low MLP regions which can't be exploited (6.8% and 5.4% for SPEC2000 and SPEC2006). This reduction in instructions also translates to a slight reduction in power (about 1% across benchmarks). This result yields an important message. If an MLP technique is expensive to implement, then select a customized core and only implement it there. On the other hand, if an MLP technique is cheap, add it everywhere. In either case, you get the best performance due to our effective hardware scheduling mechanism which exploits core-level differences.

5.6. SCE Execution Analysis

Table IV displays details on the switching overhead for our SCE proposal. The *Overhead* column indicates the overhead due to switching cores. Our scheduling mechanism can overlap most of the context switch with L2 misses when switching between *Core4* to *CLP+VP*. This explains why, out of the total overhead, the migration to the MLP core contributes an average of 24%. 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.

Table IV. Characterizing SCE: Total Overhead of Switching Over Entire Execution, Scheduling Accuracy and Value Predictor Accuracy

Benchmark	Total Overhead	Sched. Acc.	Value Pred Acc.
ammp	0.31%	96.13%	24.75%
applu	0.02%	82.43%	34.76%
bzip2	0.02%	86.67%	9.94%
crafty	0.51%	82.87%	78.55%
equake	1.13%	96.80%	47.10%
gap	1.12%	95.78%	90.91%
gzip	0.07%	96.85%	4.04%
mcf	1.07%	99.78%	49.09%
mesa	1.42%	89.86%	88.01%
mgrid	0.02%	74.08%	15.13%
parser	1.98%	93.83%	74.88%
swim	0.05%	91.45%	3.70%
twolf	0.74%	94.43%	44.31%
vortex	0.36%	87.98%	59.88%
vpr	0.91%	95.95%	22.22%
wupwise	1.78%	85.00%	44.00%
perlbench	0.02%	90.00%	12.58%
gcc	0.97%	94.51%	48.84%
milc	1.43%	93.34%	57.47%
gromacs	0.28%	83.10%	46.81%
gobmk	0.86%	91.43%	68.33%
namd	0.35%	89.86%	44.40%
dealII	1.89%	77.69%	31.33%
soplex	0.02%	97.78%	60.56%
omnetpp	0.37%	91.20%	19.09%

We can study how sensitive our design is to switching cost by varying the penalty associated with switching cores. We re-ran our experiments using a switching overhead 2.5x and 5x bigger than our original penalty (as shown in Table II). We observed a degradation in performance of 1.11% and 2.9%, respectively, for SPEC2000. For SPEC2006, we observed a 1.4% and 3.7% degradation on average, respectively. This shows that although our scheduler is able to switch cores aggressively this doesn't occur continuously. The applications which get penalized the most are the ones with the higher total overhead in Table IV.

Table IV also displays the accuracy of our scheduler and the value predictor for our *CLP+VP*. Overall our scheduling mechanism's accuracy is high. Value prediction accuracy is also interesting since low accuracy does not necessarily translate to low performance. Even for high MLP regions, when the value prediction is wrong, useful prefetches can generate enough MLP to improve performance during the re-execution of instructions. Finally, Table V shows the time spent on *Core4* and *Core2+CLP+VP* in our SCE design. As we can see here, benchmarks with huge gains spend the most time on the MLP core.

5.7. MLP Region Scheduling Benefits on *CLP+VP*

MLP region scheduling improves the performance of *CLP+VP* itself. We performed a study comparing a single core with two operation modes (MLP and non-MLP) which uses the MLP region scheduling to switch between the two modes. Overall, SCE scheduling provides a 0.6% performance improvement over the unmodified *CLP+VP* proposal. Furthermore, the increase in executed instructions due to *CLP+VP* is shown in Figure 14. As we can see, the number of increased instructions executed is decreased by 2.9% and 2.8% across SPEC2000, and SPEC2006 respectively improving the energy delay². This decrease comes from turning off MLP speculation in regions of little to no

Table V. Characterizing Execution in SCE
Mode: Percent of Execution Spent on *Core4*
and *Core2+CLP+VP* in SCE Mode

Benchmark	Core4	Core2+CLP+VP
ammp	32%	68%
applu	78%	22%
bzip2	92%	8%
crafty	78%	22%
equake	07%	93%
gap	23%	77%
gzip	06%	94%
mcf	05%	95%
mesa	83%	17%
mgrid	72%	28%
parser	48%	52%
swim	12%	88%
twolf	11%	89%
vortex	83%	17%
vpr	90%	10%
wupwise	11%	89%
perlbench	47%	53%
gcc	12%	88%
milc	26%	74%
gromacs	58%	42%
gobmk	81%	19%
namd	69%	31%
dealII	92%	8%
soplex	24%	76%
omnetpp	46%	54%

Fig. 14. Increase in executed instructions from *CLP+VP* vs benefit of MLP region scheduling.

MLP. This decrease in executed instructions benefits applications with low prediction accuracy (by up to 10%). On the other hand it also reduces performance (by up to 2%) for applications with high prediction accuracy since we speculate less.

6. RELATED WORK

6.1. Memory Level Parallelism

Although we are in the era of CMPs, sequential performance is important. Lots of research has focused on designing processors that attack the memory wall by increasing the program's MLP. Prefetching in the form of helper threads is a technique used to extract MLP [Collins et al. 2001; Moshovos et al. 2001; Ganusov and Burtcher 2006]. Execution of the helper thread is done in-parallel on a SMT, or on a separate core for CMP architectures. Other techniques such as WIB [Lebeck et al. 2002], CFP [Srinivasan et al. 2004], KILO [Cristal et al. 2004], and BOLT [Hilton and Roth 2010]

focus on increasing the window size by unblocking the pipeline on cache misses, thereby increasing the MLP. 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 [Mutlu et al. 2003; Dundas and Mudge 1997], CAVA [Ceze et al. 2006], and Clear [Kirman et al. 2005] 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. When the memory request returns it re-executes the instructions or if the value was correct, it continues execution. Chou et al. [2004] 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 extracting 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 [Srinivasan et al. 2004] 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 Multicore Processors

Asymmetric Multicore Processor designs have been proposed as a solution to achieve higher performance per watt ratio for executing a wider range of applications [Kumar et al. 2004; Becchi and Crowley 2006; Suleman et al. 2009]. This does not always result in improved performance over a homogeneous system for single threaded applications. Given the same area, previous proposals [Suleman et al. 2009; Becchi and Crowley 2006; Kumar et al. 2004; Saez et al. 2010] 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. [2007]. 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 we use fully functional cores which can work independently if needed. Execution is also always on one core rather than spread across cores.

Other recent AMP proposals use a slipstream [Najaf-abadi and Rotenberg 2009; Zhou 2005] inspired design to speedup sequential performance. These techniques however require execution of the same application on multiple cores concurrently, or require recompilation of the applications to create a reduced version of the program that will work as a perfect predictor or prefetcher [Garg and Huang 2008].

7. CONCLUSION

Due to power limitations, chips are having larger portions of the chip powered off. Asymmetric Multicore Processors (AMPs) offer a unique opportunity to utilize this powered off chip real estate and customize cores for specific regions of execution. In this study, we focus on regions of execution dominated by high MLP. 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. Therefore, we conclude that customization for MLP is important on an AMP.

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 benefits of other cores during regions with little to no MLP. Using SCE, we achieve performance improvements of 5.3% and 6.6% for SPEC2000 and SPEC2006, respectively, with a maximum speedup of 14.5%. For the same study, it achieves a 18.3% and 21.1% energy delay² reduction for SPEC2000 and SPEC2006, respectively.

Our findings yield an important message for hardware designers. Core customization enables efficient exploitation of MLP. Even if our MLP technique is cheap enough to add to all cores, all of the performance can be obtained on a single customized core. Also, customization leads to power efficiency. Finally, application steering mechanisms for MLP are simple and effective. As we enter the era of dark silicon and consider techniques that exploit customization to boost single thread performance, better exploitation of MLP is feasible and beneficial.

REFERENCES

- BECCHI, M. AND CROWLEY, P. 2006. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the ACM International Conference on Computing Frontiers*.
- CEZE, L., STRAUSS, K., TUCK, J., TORRELLAS, J., AND RENAU, J. 2006. CAVA: Using checkpoint-assisted value prediction to hide L2 misses. *ACM Trans. Architect. Code Optim.*
- CHOU, Y., FAHS, B., AND ABRAHAM, S. 2004. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the ACM International Symposium on Computer Architecture*.
- CHOUDHARY, N. K., WADHAVKAR, S. V., SHAH, T. A., MAYUKH, H., GANDHI, J., DWIEL, B. H., NAVADA, S., NAJAF-ABADI, H. H., AND ROTENBERG, E. 2011. FabScalar: Composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template. In *Proceedings of the ACM International Symposium on Computer Architecture*.
- COLLINS, J. D., TULLSEN, D. M., WANG, H., AND SHEN, J. P. 2001. Dynamic speculative precomputation. In *Proceedings of the IEEE International Symposium on Microarchitecture*.
- CRISTAL, A., SANTANA, O. J., VALERO, M., AND MARTÍNEZ, J. F. 2004. Toward kilo-instruction processors. *ACM Trans. Architect. Code Optim.*
- DUNDAS, J. AND MUDGE, T. 1997. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the ACM International Conference on Supercomputing*.
- ESMAEILZADEH, H., BLEM, E., ST. AMANT, R., SANKARALINGAM, K., AND BURGER, D. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th ACM International Symposium on Computer Architecture, ISCA '11*.
- GANUSOV, I. AND BURTSCHER, M. 2006. Future execution: A prefetching mechanism that uses multiple cores to speed up single threads. *ACM Trans. Architect. and Code Optim.*
- GARG, A. AND HUANG, M. C. 2008. A performance-correctness explicitly-decoupled architecture. In *Proceedings of the IEEE International Symposium on Microarchitecture*.
- gcc 2009. GNU compiler collection. <http://gcc.gnu.org/>.
- HILTON, A. AND ROTH, A. 2010. BOLT: energy-efficient out-of-order latency-tolerant execution. In *Proceedings of the IEEE International Symposium of High Performance Computer Architecture*.
- KIRMAN, N., KIRMAN, M., CHAUDHURI, M., AND MARTINEZ, J. F. 2005. Checkpointed early load retirement. In *Proceedings of the IEEE International Symposium of High Performance Computer Architecture*.
- KUMAR, R., TULLSEN, D. M., RANGANATHAN, P., JOUPPI, N. P., AND FARKAS, K. I. 2004. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the ACM International Symposium on Computer Architecture*.
- LEBECK, A. R., KOPPANALIL, J., LI, T., PATWARDHAN, J., AND ROTENBERG, E. 2002. A large, fast instruction window for tolerating cache misses. In *Proceedings of the ACM International Symposium on Computer Architecture*.
- MOSHOVOS, A., PNEVMATIKATOS, D. N., AND BANIASADI, A. 2001. Slice-processors: An implementation of operation-based prediction. In *Proceedings of the ACM International Conference on Supercomputing*.

- MUTLU, O., KIM, H., AND PATT, Y. N. 2006. Efficient runahead execution: Power-efficient memory latency tolerance. In *Proceedings of the IEEE International Symposium on Microarchitecture*.
- MUTLU, O., STARK, J., WILKERSON, C., AND PATT, Y. N. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the IEEE International Symposium of High Performance Computer Architecture*.
- NAJAF-ABADI, H. H. AND ROTENBERG, E. 2009. Architectural contesting. In *Proceedings of the IEEE International Symposium of High Performance Computer Architecture*.
- PALACHARLA, S., JOUPPI, N. P., AND SMITH, J. E. 1997. Complexity-effective superscalar processors. In *Proceedings of the IEEE International Symposium of High Performance Computer Architecture*.
- PERICAS, M., CRISTAL, A., CAZORLA, F. J., GONZALEZ, R., JIMENEZ, D. A., AND VALERO, M. 2007. A flexible heterogeneous multi-core architecture. In *Proceedings of the IEEE International Conference on Parallel Architecture and Compilation Techniques*.
- RENAU, J., FRAGUELA, B., TUCK, J., LIU, W., PRVULOVIC, M., CEZE, L., SARANGI, S., SACK, P., STRAUSS, K., AND MONTESINOS, P. 2005. SESC simulator. <http://sesc.sourceforge.net>.
- SAEZ, J. C., PRIETO, M., FEDOROVA, A., AND BLAGODUROV, S. 2010. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the ACM International Conference on Computer Systems*.
- SHERWOOD, T., PERELMAN, E., HAMERLY, G., SAIR, S., AND CALDER, B. 2003. Discovering and exploiting program phases. *IEEE Micro*.
- SRINIVASAN, S. T., RAJWAR, R., AKKARY, H., GANDHI, A., AND UPTON, M. 2004. Continual flow pipelines. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
- STINE, J., CASTELLANOS, I., WOOD, M., HENSON, J., LOVE, F., DAVIS, W., FRANZON, P., BUCHER, M., BASAVARAJAIAH, S., OH, J., AND JENKAL, R. 2007. FreePDK: An open-source variation-aware design kit. In *Proceedings of the IEEE International Conference on Microelectronic Systems Education*.
- SULEMAN, M. A., MUTLU, O., QURESHI, M. K., AND PATT, Y. N. 2009. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
- WECHSLER, O. 2006. Inside Intel(R) Core(TM) microarchitecture setting new standards for energy-efficient performance. In *Tech. rep.*. <http://www.intel.com/technology/architecture-silicon/core/>.
- ZHOU, H. 2005. Dual-Core execution: Building a highly scalable single-thread instruction window. In *Proceedings of the IEEE International Conference on Parallel Architecture and Compilation Techniques*.
- ZHOU, H. AND CONTE, T. M. 2003. Enhancing memory level parallelism via recovery-free value prediction. In *Proceedings of the IEEE International Conference on Supercomputing*.

Received July 2011; revised October 2011; accepted November 2011