Hybrid Multi-Core Architecture for Boosting Single-Threaded Performance

Jun Yan, Wei Zhang
Department of Electrical and Computer Engineering
Southern Illinois University Carbondale
Carbondale, IL 62901
{jun,zhang}@engr.siu.edu

Abstract

The scaling of technology and the diminishing return of complicated uniprocessors have driven the industry towards multi-core processors. While multithreaded applications can naturally leverage the enhanced throughput of multi-core processors, a large number of important applications are single-threaded, which cannot automatically harness the potential of multi-core processors. In this paper, we propose a compiler-driven heterogeneous multi-core architecture, consisting of tightly-integrated VLIW (Very Long Instruction Word) and superscalar processors on a single chip, to automatically boost the performance of single-threaded applications without compromising the capability to support multithreaded programs. In the proposed multi-core architecture, while the high-performance VLIW core is used to run code segments with high instruction-level parallelism (ILP) extracted by the compiler; the superscalar core can be exploited to deal with the runtime events that are typically difficult for the VLIW core to handle, such as L2 cache misses. Our initial experimental results by running the pre-execution thread on the superscalar core to mitigate the L2 cache misses of the main thread on the VLIW core indicate that the proposed VLIW/superscalar multi-core processor can automatically improve the performance of single-threaded general-purpose applications by up to 40.8%.

1. Introduction

With the diminishing return of complex uniprocessors, computer industry is rapidly moving towards single-chip multi-core processors or chip multiprocessors (CMP). These multi-core processors are typically built by replicating two or more independent microprocessors (cores) onto a single chip, which are also called homogeneous multi-core processors in this paper. The major benefits of homogeneous CMPs include reduced design complexity, reusability, and high throughput. While these multi-core processors can *naturally* improve the throughput for multi-programmed and multithreaded code; the latency of a single-threaded program (or each thread of multithreaded programs) is not changed. Unfortunately, many important applications are written in sequential programming languages, such as C or C++, which generally assume a single thread execution model. Therefore, these single-threaded applications cannot benefit directly from these CMPs. Therefore, it is a necessity to develop techniques for multi-core processors to reduce the latency of single-threaded applications while maintaining the advantage of providing higher throughput for multithreaded software.

Single-threaded applications can benefit from CMPs if the compiler can automatically parallelize those applications so that multiple tasks can run simultaneously on different cores of CMPs. However, decades of research in parallel compilation has only yielded limited success [12]. While scientific applications with regular data accesses and predictable control flow can be successfully parallelized [5], compiler-based parallelization of general-purpose programs with irregular data access patterns and less-predictable control flow has been much less effective [6]. Researchers then turned to investigate speculative techniques to extract parallelism more aggressively [7, 8, 9, 10]. Nevertheless, these techniques generally require significant hardware support to cope with mis-speculations, which can increase the hardware complexity of CMPs substantially. Moreover, the success of these speculative techniques is highly dependent on the aggressiveness and the accuracy of speculation. While conservative speculative techniques typically have very limited capability to increase performance beyond what the non-speculative techniques can achieve; too aggressive speculation may lead to excessive mis-speculations, which will not only compromise performance, but also waste energy.

Another approach to achieving both high throughput and low latency for CMPs is to use the single-ISA heterogeneous multi-core processors, proposed by Kumar et al. [1]. This single-ISA heterogeneous multi-core architecture is composed of cores of varying sizes, performance and complexity, but with the same ISA. In this architecture, the single-threaded programs can exploit powerful cores (i.e., complex superscalar processors) for achieving low latency without re-writing or re-compiling the programs. On the other hand, the multi-programmed or multithreaded applications can be hosted on many simple cores for higher throughput. However, to attain the single-threaded performance that is scalable and comparable to the performance of the state-of-the-art uniprocessors, the most advanced (and the most complex) uniprocessor must be incorporated into the multi-core die. Therefore, while the single-ISA heterogeneous multi-core design can provide a near-term solution for balancing latency and throughput, we believe that in the long run, the single-threaded performance of CMPs cannot solely rely on the existence and integration of very powerful and scalable uniprocessors. Instead, our insight is that the complexity of the complicated single-core or multi-core hardware should be mostly or partly shifted to the compiler in order to improve both single-threaded and multithreaded performance of multi-core computing without having to fundamentally re-writing the applications.

In this paper, we propose a compiler-friendly multi-ISA heterogeneous multi-core architecture (also called the VLIW/superscalar heterogeneous multi-core architecture, hybrid multi-core processor, hybrid multiprocessor, hybrid architecture, or heterogeneous-ISA multi-core architecture in this paper) to enable the compiler to play a more important role to enhance both the single-threaded and multithreaded performance for multi-core applications. In contrast to single-ISA heterogeneous multi-core processors [1] that consists of cores with different microarchitecture but the same architecture, the proposed hybrid multi-core architecture allows different cores to have not only varying microarchitectures, but also different architectures (i.e., VLIW and superscalar cores) in order to achieve the best design tradeoffs between performance and hardware/software complexity. The proposed hybrid multi-core architecture can leverage the advanced compiler optimizations for VLIW processors, while complementing the disadvantages of statically-scheduled VLIW core(s) with the dynamic capability of superscalar core(s) *to automatically boost the performance of single-threaded applications without compromising the flexibility to benefit multithreaded programs*. In addition to the prospective performance benefits, the hybrid architecture can also exploit the heterogeneous ISAs to potentially reduce energy dissipation by maximizing the utilization of the energy-efficient VLIW core(s). Moreover, with effective compiler support (also called hybrid compiler in this paper) and the reuse of existing VLIW and superscalar design, the proposed hybrid multi-core architecture does not increase the hardware complexity significantly or compromise the programmability.

The rest of this paper is organized as follows. In Section 2, we introduce the architecture, compiler support, and the potential major benefits of the proposed VLIW/superscalar heterogeneous multi-core processors. Section 3 details the idea of running pre-execution code on the superscalar core for mitigating the L2 cache misses of the VLIW core through the shared L2 cache, which is our first step to exploring the feasibility and the performance potential of the proposed hybrid multi-core architecture. Section 4 explains the evaluation methodology and Section 5 gives the initial experimental results. The related work is discussed in Section 6. Finally, we make conclusions and show future work direction in Section 7.

2. The VLIW/Superscalar Heterogeneous Multi-Core Processor

2.1 Background

Superscalar and VLIW are two major types of ILP processors [1]. While superscalar processors employ complex hardware to extract instruction-level parallelism dynamically, VLIW machines rely on advanced compilers to discover ILP statically by keeping hardware simple. Consequently, superscalar processors can transparently increase the performance without re-compiling the programs. Moreover, superscalar processors have the capability to efficiently deal with runtime events, such as branch direction and target address, load latencies (cache hit/miss), and memory dependencies, which are typically less predictable at the compilation time. In contrast, VLIW processors generally cannot issue instructions dynamically [2], thus runtime events such as cache misses will cause the VLIW pipelines to stall.

On the other hand, although the superscalar processors have the advantage of scheduling instructions dynamically, the scope of the analysis and the degree of ILP extracted at runtime are typically limited by the size of the hardware instruction window. Also, to support dynamic scheduling, the hardware complexity of superscalar processors has become overwhelming, which may compromise the clock frequency, as well as increase the design complexity and cost significantly. By comparison, VLIW machines can shift major hardware complexity to optimizing compilers, which are capable of analyzing larger scopes of instructions and transforming the programs to potentially extract higher ILP. More importantly, because the complex control hardware responsible for operation decomposition is eliminated from the critical path, the VLIW processors are able to achieve maximum performance by enabling high clock frequency while still exploiting available instruction-level parallelism in the code [29]. Furthermore, since the VLIW hardware does not need to re-do the work done by the compiler (e.g., analyzing data dependences, extracting and scheduling independent instructions), VLIW machines can perform more energy-efficient computing. Because of these advantages, VLIW architectures have been increasingly used in high-performance and power-efficient embedded and DSP processors [24, 25].

While both VLIW and superscalar processors have their strengths and weaknesses; with the advent of multi-core computing era, it becomes feasible and attractive to combine their advantages while avoiding their respective disadvantages for supporting a variety of workloads that demand both low latency and high throughput. Additionally, this heterogeneous-ISA multi-core architecture can potentially explore larger design space (compared to single-ISA multi-core) to find better tradeoffs between hardware and software complexity for multi-core computing.

2.2 Architecture

In this section, we present a high-level overview of the hybrid multi-core architecture, which is composed of both VLIW core(s) and superscalar core(s). The goal of this hybrid multi-core architecture is to leverage and balance both the compiler and hardware techniques to cooperatively and automatically improve the performance of single-threaded applications while maintaining the flexibility to support multithreaded programs. While the proposed architecture can contain various number of VLIW and superscalar processors with different complexity, area, performance, as well as different interconnections; the extensive design space exploration of the proposed multi-core architecture is out of the scope of this paper. For simplicity, this paper concentrates on studying a hybrid dual-core processor, which is composed of a VLIW core and a superscalar core, as depicted in Figure 1. As we can see from Figure 1, a given single-threaded program will be analyzed and automatically parallelized by the compiler before running on the hybrid multi-core (see subsection 2.4 for detail of the compiler support). For a dual-core design, two threads will be created. The thread with high ILP, regular control flow [14] and predictable load latencies [15] will be passed to the VLIW core, whose execution plan is statically determined by the compiler and will be enforced by the hardware at run time (note that the VLIW core will employ interlocks to ensure the correctness of execution [11]). In contrast, the thread with irregular ILP and unpredictable control flow or memory access patterns, will run on the superscalar core to take advantage of its dynamic scheduling capability. Additionally, speculative threads can also be extracted from the single-threaded applications. These speculative threads will be bound to the superscalar core, which is suitable for supporting the speculative state and the recovery of mis-speculation, with minimum hardware extension. Each core of the hybrid multi-core architecture has its own L1 instruction cache and L1 data cache. A bus-based snooping protocol is used to maintain coherence among data caches. In our current design, the VLIW core and the superscalar core communicate through a shared unified L2 cache for simplicity. In our future work, we will also consider customizing the interconnec-

[1] Explicit Parallel Instruction Computing (EPIC) is regarded an important evolution of VLIW architecture, which has also absorbed many of the best ideas of superscalar processors [11].

[2] There have been some research efforts to enhance dynamic issue in VLIW machines [45, 46]; however, the major disadvantage of this approach is that it increases the VLIW hardware complexity and may compromise the clock cycle time if the rescheduling hardware is on the critical path.

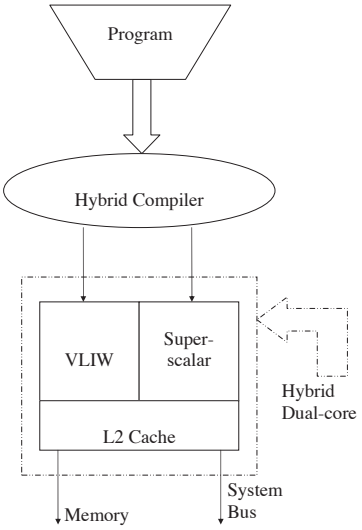

Figure 1. A high-level overview of the VLIW/superscalar dual-core architecture.

tions, such as adding a small operand transfer buffer between the VLIW and superscalar cores for finer-grain inter-core communications at the cost of increased design complexity. Such a more tightly-coupled hybrid dual-core will enable the hybrid compiler to partition threads in a finer granularity to better balance workloads for achieving even higher performance (see subsection 2.4 for detail).

2.3 Potential Benefits

The proposed hybrid multi-core architecture allows the VLIW processor and the superscalar processor to complement each other for combining their advantages while avoiding their disadvantages. Qualitatively speaking, the potential benefits include the following:

1. Due to the overwhelming complexity of superscalar design and the relatively narrow scope of runtime analysis, the clock frequency and the degree of parallelism supported by the superscalar microprocessors are fundamentally limited. In the proposed hybrid multi-core architecture, however, the VLIW core can operate at very aggressive clock rates and provide a large number of functional units, which can be used to support threads of higher ILP for better single-threaded performance, with the assistance of the compiler.
2. Due to the existence of the high-performance VLIW core in the hybrid architecture, the superscalar core does not need to be very wide or complicated ,as compared to the largest core of the single-ISA heterogeneous multi-core architecture [1], which can improve the energy efficiency of the superscalar core.
3. Compared with very wide VLIW processors or multi-clustered VLIW processors, the hybrid multi-core architecture can exploit the superscalar core to better cope with the runtime events or speculative operations. This will remove the burden of VLIW compilers to conservatively analyzing and transforming the programs, thus enabling more aggressive compiler optimizations for achieving higher performance. For instance, if the superscalar core can perfectly prefetch the data into the shared L2 cache for applications running on the VLIW core, the VLIW compiler can schedule operations on the load dependence chain earlier without having to attempt to statically tolerate the possible L2 misses.
4. Compared with homogeneous or single-ISA heterogeneous multi-core processors, typically consisting of pure superscalar cores, the proposed hybrid multi-core architecture is more friendly to compilers, making it possible to leverage the compiler technology to extract both non-speculative and speculative threads, as well as multi-grain parallelism to achieve higher single-threaded performance without increasing the hardware complexity substantially. Compared with hardware-centric approach, the compiler can perform large-scope analysis (i.e., procedures or programs), transform the code and data layout to extract more parallelism at different granularities. In addition, the proposed hybrid architecture can also exploit the superscalar cores to handle dynamic events effectively.
5. The hybrid multi-core architecture can also be potentially more energy efficient than the homogeneous or single-ISA heterogeneous multi-core architectures; since the work done by the VLIW compiler does not need to be repeated by the VLIW hardware. By comparison, the superscalar core has to perform complicated runtime analysis by hardware for achieving comparable performance, which is energy-hungry.
6. The proposed hybrid architecture can also be potentially very useful for high-performance embedded applications. In general, VLIW processors are much easier to scale and customize than superscalars, in that they consist of regular functional units and register bank components, rather than the more ad hoc and difficult-to-design superscalar control units [25]. Therefore, one can exploit domain or application specific knowledge to tailor the VLIW core of a hybrid multi-core processor to maximize the performance per watt.

However, compared with homogeneous or single-ISA heterogeneous multi-core processors, the hybrid architecture consists of processors with different ISAs, which will increase the compiler complexity and demand ISA-specific code generations and optimizations in the backend. Also, another major advantage of single-ISA heterogeneous multi-core architecture [1] is that it can enhance single-threaded performance transparently without recompilation, which is indeed required in the hybrid architecture. Nevertheless, we argue that increasing the compiler complexity is the penalty we need to pay for decreasing the complexity of the multi-core hardware, unless there are breakthroughs in new multi-core programming paradigms so that programmers can develop multithreaded applications with reasonable efforts. Also, with the prevalence of multi-core processor based computing platforms, the compilation time may become a less important concern.

2.4 The Proposed Compiler Support

The key to the success of the proposed hybrid multi-core architecture is the design and implementation of the hybrid compiler that can effectively partition the single-threaded programs and extract both coarse-grain and fine-grain parallelism. Compared to the single-ISA heterogeneous multi-core architecture [1] that relies on complex superscalar core to reduce latency, where compilers only play a limited role, the hybrid architecture is more friendly to compilers. At the thread-level parallelism (TLP), the hybrid compiler is aware of the execution bandwidths and frequencies of both the VLIW and superscalar cores, as well as the inter-core communication latency, based on which the compiler can extract cooperative threads, either non-speculative or speculative, to harness the full potential of multi-core processors. At the instruction-level parallelism, while the microarchitecture of the superscalar core is invisible to the compiler, the architectural details of the VLIW core, such as the number of functional units, the latencies of operations, etc., are exposed to the compiler for exploring high ILP.

The compiler-friendliness of the hybrid architecture allows the hybrid compiler to potentially leverage decades of research efforts in VLIW/EPIC compilers, such as IMPACT [33], trimaran [26] and ORC compiler [47], as well as recent progress in compiler support for thread partitioning [2, 34]. More specifically, the proposed hybrid multi-core architecture enables the construction of powerful compilers to boost single-threaded performance automatically due to the following factors:

1. Both non-speculative and speculative threads are supported by the hybrid architecture. While the non-speculative threads can harness both VLIW and superscalar cores (where the VLIW core will play a major role), the speculative threads can be naturally supported by the superscalar core, which can be used as helper threads (e.g., pre-execution, value prediction, etc.) [21] to boost the performance of the main thread.
2. In addition to thread-level parallelism, the hybrid compiler can take advantage of the high-performance VLIW core to perform aggressive ILP optimizations, such as software pipelining [39], trace [40], superblock [41] or hyperblock [42] scheduling to exploit high instruction level parallelism for better performance. With the support of helper threads on the superscalar core that is tightly integrated with the VLIW core, these compiler optimizations can be potentially carried out more aggressively for even higher performance.
3. The compiler, in conjunction with the operating system, can exploit the heterogeneity of the proposed hybrid multi-core architecture to improve the performance per watt by matching threads with various properties to different cores that can execute those threads most energy-efficiently.

More specifically, the backend of the hybrid compiler needs to incorporate new optimization phases to partition programs into profitable threads, either speculative or non-speculative, in order to harness the potential of the hybrid multi-core architecture. We plan to develop these hybrid multi-core specific optimizations in two major phases as listed below:

Phase 1: pre-execution thread extraction. Since the memory wall problem is one of the most significant obstacles to single-threaded performance, our first optimization is to extract the pre-execution thread to run on the superscalar core for minimizing the L2 cache stalls of the VLIW core, which is discussed in detail in Section 3.

Phase 2: non-speculative thread partitioning. While phase 1 focuses on exploiting speculative threads or helper threads to boost the performance of the main thread, the second phase will center on extracting non-speculative multi-grain parallelism for further performance improvement. The hybrid backend compiler can perform control-flow and data-flow analyses [13] to partition the source code into two parts: predictable code regions [14, 15] and unpredictable code regions, where each region can be a set of procedures, loops or basic blocks [16]. The predictable code regions comprise the code whose runtime branch behavior [14] and load latencies [15] are highly predictable by using either static analysis or profiling. These predictable code regions will then be executed on the high-performance VLIW core. In contrast, the unpredictable code can be better handled by the superscalar core. After this initial partitioning, the backend compiler then schedules the instructions in each region and estimates the execution time by exploiting the profiling information. Based on the synchronization cost and the estimated execution latency, the hybrid compiler then conducts profitability analysis for the initial partition. If the workload is not well balanced between the VLIW and superscalar cores, the initial regions, either predictable or unpredictable, will be re-partitioned into sub-regions with finer granularities, which will then be re-scheduled to the other core. At the same time, the synchronization instructions will be inserted into the sub-regions to guarantee the correctness. This re-partitioning step continues until no further benefit can be obtained. Next, for each region or sub-region (micro-thread) assigned to a specific core, the compiler will carry out instruction selection specifically for the target core, and perform ILP optimizations [13], such as instruction scheduling and register allocation, to extract the maximum instruction-level parallelism supported by the underlying cores. For superscalar core(s) in particular, the compiler will focus on placing the remotely independent instructions physically closer so that they can be executed in parallel by the superscalar processors at runtime. When the pre-execution optimization is enabled, the compiler also needs to merge the pre-execution instructions and the non-speculative ILP instructions into a single thread for the superscalar core. Due to the significant impact of L2 stalls on the performance, the priority will be given to scheduling the pre-execution instructions as early as possible to minimize the L2 miss stalls.

Currently, we have reached an implementation of phase 1.

3. Initial Quantitative Study: Running the Pre-execution Thread on the Superscalar Core to Mitigate L2 Misses for the VLIW Core

The extensive exploration of the hybrid multi-core architectural design space and the hybrid compiler optimization space is out of the scope of this paper. As the first step, this paper attempts to investigate the potential of the hybrid dual-core architecture in a timely fashion by running the compiler-extracted pre-execution thread on the superscalar core in order to mitigate the L2 cache misses of the VLIW core. However, it should be noted that the performance potential of the proposed hybrid multi-core architecture is not limited to the initial work conducted in this paper on exploiting pre-execution thread only.

Pre-execution is not a new idea. Researchers have studied the use of pre-execution thread (i.e., helper thread) to improve the performance of the main thread on multithreaded processors [17, 18, 19], as well as on homogeneous chip multiprocessors [20, 21]. By comparison, this work makes use of the superscalar core to run the pre-execution thread for minimizing the L2 miss stalls for the VLIW core, within the framework of the dual multi-core architecture. It should be noted that our preliminary performance evaluation at this stage aims at providing some insights on the feasibility and the performance potential of the proposed hybrid architecture in a timely fashion, rather than exploring superior pre-execution mechanisms than prior work [17, 18, 19].

In general, VLIW processors have very limited capability to tolerate caches misses, especially for L2 cache misses. Traditionally, VLIW compilers have to assume that all the load operations will hit in the L1 cache and schedule load-dependent instructions accordingly [15]. However, if the load operations actually miss in the L1 or even the L2 cache at runtime, the VLIW instruction pipelines have to be stalled until loads return to guarantee the correctness; although there may be some other load-independent instructions that can be executed during the load stall cycles. Alternatively, VLIW compilers can assume that all loads will miss in the runtime; however, this scheme is not feasible for most applications unless considerable instruction-level parallelism can be extracted from the programs. Recent research effort indicates that load latencies are predictable for many applications based on cache profiling [15]. Particularly, a small number of load operations are responsible for the majority of data cache misses, which are called *delinquent loads*. Based on this observation, Abraham et al. proposed to schedule load-dependent operations based on the predicted load latencies [15], which is also used in the programmatic cache hierarchy management of EPIC architectures [11]. While this load-latency-aware scheduling can achieve better performance than the all-hit scheduling, it may be less successful for long load latencies, such as L2 cache misses. The reason is that it requires huge ILP available that is independent on

the loads, which are very likely on the critical path. Prefetching can somehow mitigate the cache miss problem; however, data prefetching is generally less effective for programs with complex array subscripts and pointer-chasing code. With a tightly-coupled superscalar core available in the hybrid dual-core processor, pre-execution can be employed to minimize the L2 cache stalls for the VLIW core. As a result, the VLIW compiler can schedule instructions more aggressively for maximizing performance, without worrying too much about the runtime L2 cache misses. In addition, this paper studies an approach to combine data prefetching to minimize L1 cache misses and the pre-execution to decrease L2 cache misses, which can further improve the single-threaded performance.

In this paper, we employ cache profiling to identify delinquent loads. More specifically, we perform simulation to record the number of stall cycles caused by each static load instruction in the application, and identify the top L2-cache-missing loads whose load stall cycles are larger than a pre-set threshold (also called *delinquent loads threshold* in this paper). By default, the threshold is set to be 10%, implying that only load operations whose accumulated stall cycles is more than 10% of the total stall cycles of the application will be identified as delinquent loads. It should be noted that this threshold is implemented as a parameter in our simulator and can be varied for studying sensitivity. Since our cache profiling experiments have only returned a few delinquent loads (which is in accord with previous work [15]), we manually extract the load dependence chains (i.e., the program slice leading to the memory address calculation) for those delinquent loads by examining the program dependence graph. We then insert those instructions into the pre-execution thread, which will be executed on the superscalar core. In the pre-execution thread, all load operations are converted into non-faulting loads to avoid exceptions. Moreover, all store operations in the pre-execution thread are removed to ensure the semantic correctness of the main thread on the VLIW core.

4. Evaluation Methodology

In order to test the idea of cooperative exploitation of pre-execution thread by heterogeneous cores for improving single-threaded performance, we try to leverage existing VLIW and superscalar architectures and simulators; although the proposed hybrid multi-core architecture is not limited to the initial simulation framework studied in this paper. More specifically, in our simulation framework, the VLIW core is based on the HPL-PD 1.1 architecture [22], and the superscalar core is similar to the Alpha 21264 processor [23]. The configurations of these two cores are given in Table 1. The hybrid compiler is built upon the Trimaran compiler framework [26], which provides an optimizing backend compiler (i.e., elcor), an extensible intermediate representation (IR), and a machine description facility to target different architectures. We develop a cycle-accurate model of the hybrid dual-core, based on the integration of the VLIW simulator from Trimaran [26] and the superscalar simulator – simplescalar [27]. To evaluate the performance potential of the proposed hybrid dual-core architecture, we select seven single-threaded applications (that can be successfully compiled and run on our experimental framework at present) from SPEC 2000 INT and SPEC 92 FP benchmark suites [28]. The salient behavior of those benchmarks are shown in Table 2.

4.1 Schemes Under Consideration

In our experiments, we study the performance of the following schemes:

1. `Base:` This is the scheme that only uses the default VLIW processor (equivalent to the VLIW core of the hybrid dual-core processor) to run applications. The default configuration of this VLIW processor is specified in Table 1. It should be noted that at this stage, our initial performance evaluation aims at providing some insights on the interactions of the superscalar and VLIW cores to enhance single-threaded performance. In our future work (after the implementation of the hybrid compiler), we will provide quantitative performance comparison between the proposed architecture and the base performance of a VLIW processor, a superscalar processor, and a single-ISA heterogeneous multi-core processor [1] with the same area.
2. `Pre-execution (P):` This is the scheme of the hybrid dual-core processor, where the superscalar core is exploited to run the pre-execution thread for reducing the L2 stalls for the VLIW core.
3. `Pre-execution and Prefetch (PP):` This scheme combines the pre-execution scheme with the data prefetching on the VLIW core.

   It should be noted that when we make sensitivity experiments on the *delinquent loads threshold*, we use the notation P-T% or PP-T% to represent the P (Pre-execution) scheme and the PP (Pre-execution and Prefetch) scheme with different thresholds (T). For instance, P-10% means the Pre-execution scheme with the *delinquent loads threshold* set to be 10% (see subsection 5.3 for more information).

Parameter	Dual-core Value	
Core	VLIW	Superscalar
Datapath	4IFUs, 2FPUs, 2Ld/Sts, 1BrU 64 registers	4 Issue 64 registers, 80-RUU, 40-LSQ
L1 I-cache	32K, 4-way, 64 byte block 1 cycle latency	32KB, 1-way, 32 byte blocks 1 cycle latency
L1 D-cache	32K, 4-way, 32 byte block 1 cycle latency	32KB, 1-way, 32 byte blocks 1 cycle latency
L2 cache	1MB, 8-way, 64 byte block 10 cycle latency	
Memory	100 cycle latency	

Table 1. Configuration parameters of simulated hybrid dual-core processor.

Benchmark	Source	Static Insts	Dynamic Cycles	L2 Miss Rate
164.gzip	SPECint 2000	10369	2282118441	0.02%
176.gcc	SPECint 2000	485692	5438711558	1.20%
181.mcf	SPECint 2000	3381	19924360712	4.33%
255.vortex	SPECint 2000	173611	16180635686	0.15%
256.bzip2	SPECint 2000	9184	12241634512	0.32%
btrix	SPECfp 92	1224	14747502	7.10%
vpenta	SPECfp 92	781	44034396	2.78%

Table 2. Salient behavior of selected benchmarks.

5. Experimental Results

5.1 Performance Results of Pre-execution

Our first experiment investigates the potential performance improvement of the pre-execution scheme of the hybrid dual-core, as compared to that of the base scheme (i.e., a single VLIW processor). As can be seen from Figure 2, the effect of pre-execution on the single-threaded performance varies greatly for different applications. More specifically, running the pre-execution thread on the superscalar core improves the performance of `btrix` and `181.mcf` by 40.8% and 14.1% respectively, since both benchmarks suffer from the L2 cache misses. On the other hand, there are almost no performance benefits for other SPEC 2000 INT benchmarks, including `164.gzip, 176.gcc, 255.vortex`

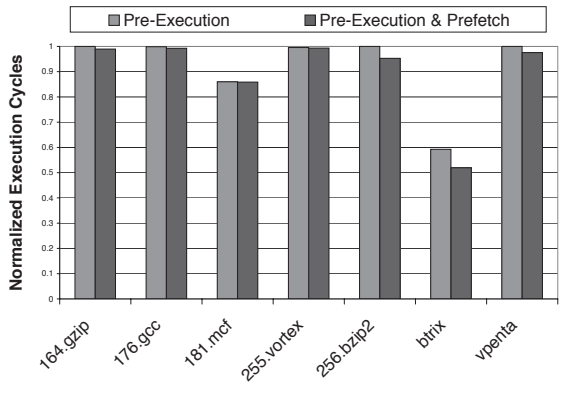

Figure 2. Execution cycles of pre-execution only, and a combination of both pre-execution and prefetching, which are normalized with respect to the base scheme.

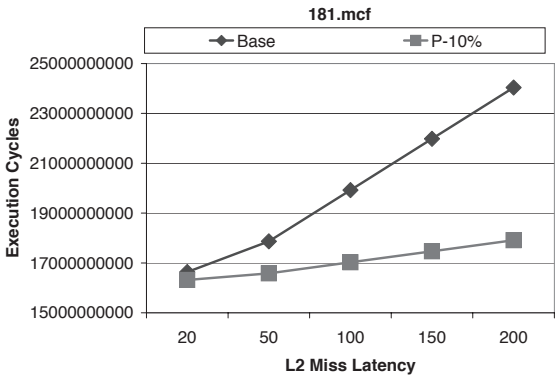

Figure 3. Execution cycles of base and pre-execution for `181.mcf` with the L2 miss latency varying from 20 cycles to 50, 100, 150 and 200 cycles.

and `256.bzip2`, since these benchmarks are not memory-intensive, and the L2 stalls are not the most severe obstacle for high performance. We also notice that while `vpenta` suffers from a moderate L2 miss rate (i.e., 2.78%), pre-execution does not have much impact on its performance. The reason is that `vpenta` only contains 2 delinquent loads, which are responsible for 22.3% of the total L2 stall cycles of the program. In contrast, `btrix` has 4 delinquent loads, which are responsible for 80.7% of the overall stall cycles, making the pre-execution more effective to boost the performance. Overall, we find that within the hybrid multi-core architecture, the superscalar core can execute the pre-execution thread to mitigate the L2 cache misses of the main thread running on the VLIW core for memory-intensive applications, leading to the improvement of single-threaded performance for these applications. Moreover, running the pre-execution thread on the superscalar core can also simplify the VLIW instruction scheduling performed by the compiler, since the pre-execution thread does not interfere with the thread running on the VLIW core and thus will not impact the scheduling.

It should be noted that while pre-execution can reduce the L2 stall cycles for some of the single-threaded applications, this is not the only optimization that can be performed in the hybrid multi-core architecture for improving single-threaded performance. Actually, our experiments indicate that the average utilization ratio of the superscalar functional units is only 4.2%. Therefore, due to the under-utilization of the superscalar core, there are still plenty of opportunities to exploit the superscalar core for further performance improvement, which will be quantitatively studied in our future research work. Moreover, in this work we assume that both the VLIW core and the superscalar core operate on the same clock frequency. In our future work, we would like to explore the potential of energy savings by varying the speed of the superscalar core, as it is currently under-utilized.

5.2 Effect of Prefetching

In the hybrid dual-core architecture, the VLIW core and the superscalar core share the L2 cache. Therefore, running the pre-execution thread on the superscalar core will have no effect on the L1 cache misses of the main thread on the VLIW core. In this paper, we also study the use of prefetching on the VLIW core to reduce the L1 cache stalls for the main thread, in addition to mitigating L2 miss stalls through the pre-execution. We have examined two different hardware-based prefetching policies, i.e., *"prefetch_always"* and *"prefetch_on_miss"*, both of which are based on *one block lookahead* (OBL) approach [30]. We find that *"prefetch_always"* returns better results on average, which is chosen as our default data prefetching policy. Figure 2 compares the execution cycles of the hybrid dual-core with pre-execution only and with a combination of pre-execution and prefetching, which are normalized with respect to the performance of the base scheme. As can be seen, prefetching can lead to additional performance enhancements for some benchmarks, such as `bzip2`, `btrix` and `vpenta`, which suffer from L1 data cache misses. For the rest of the benchmarks, however, there is very limited additional performance improvements due to prefetching. On average, a combination of pre-execution and prefetching can improve the single-threaded performance from 6.9% (of pre-execution only) to 10%.

5.3 Sensitivity Analysis

Since the pre-execution thread is used to mitigate the L2 stalls, we make experiments to study the sensitivity of the performance to different L2 cache latencies. Since many SPEC INT 2000 benchmarks are not memory-intensive, we select a representative benchmark (i.e., `181.mcf`) for this study, whose result is shown in Figure 3. To take into account the increasing gap between the processor speed and the memory speed, as well as recent proposes of providing hardware crypto support for tampering resistance and security [31, 32], which can have negative impact on the critical path of memory accesses; we vary the L2 miss latency from 20 cycles, to 50, 100, 150 and 200 cycles. The performance of pre-execution with different L2 miss latencies are compared in Figure 3. As we can see, with the increase of the L2 miss latency, the execution cycle of `181.mcf` increases significantly, indicating significant performance losses. However, by running the pre-execution thread on the superscalar core, the performance degradation with larger L2 miss latencies is reduced dramatically. Therefore, in light of the increasing memory wall problem, the pre-execution scheme studied in this paper will be an effective approach to boosting performance for future hybrid multi-core processors.

Another important parameter for pre-execution is the number of delinquent loads extracted from applications, which is controlled by the *delinquent loads threshold*. By default, the threshold is set to be 10%. In other words, only the loads that are responsible

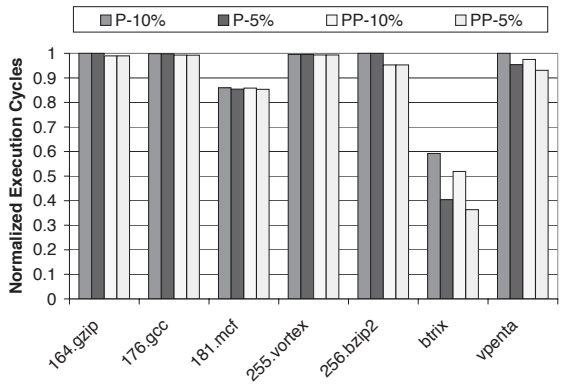

Figure 4. Execution cycles of pre-execution (P) and a combination of pre-execution and prefetching (PP) with the *delinquent loads threshold* varying from 10% to 5%, which are normalized with respect to those of the default VLIW processor.

for more than 10% of the overall cache stall cycles will be selected as delinquent loads. While pre-execution can effectively prefetch data into the L2 cache to reduce the stall cycles for those delinquent loads, other load operations can still suffer from L2 cache misses. Since our experiments indicate that the utilization of the superscalar core is very low, we can extract pre-execution threads more aggressively for a larger number of load operations. Thus, we lower the *delinquent loads threshold* to 5%, whose performance, as well as that of 10%, are presented in Figure 4. As can be seen from Figure 4, decreasing the *delinquent loads threshold* has little impact on the performance of benchmarks from SPEC INT 2000, as very few additional delinquent loads can be extracted. Moreover, since the execution time of those benchmarks is very large, those few additional delinquent loads will only have negligible impact on the overall performance. By comparison, we observe moderate performance improvements for both the P scheme and the PP scheme for loop-intensive benchmarks (e.g., `btrix` and `vpenta`). The reason is that 2, 2 and 5 *additional* delinquent loads are extracted for `btrix` and `vpenta` respectively, when the *delinquent loads threshold* is reduced to 5%. Therefore, more L2 cache stalls can be mitigated for those benchmarks.

6. Related Work

While there have been increasing research efforts on multi-core processors, most of multi-core processors proposed previously are either homogeneous [3, 4], or heterogeneous in microarchitecture but are based on a single-ISA [1, 48]. In contrast, this paper proposes a hybrid multi-core architecture by integrating VLIW processor(s) with superscalar processor(s), which have different architectures. By leveraging compiler techniques, the proposed architecture can potentially make better tradeoffs between performance and hardware/software complexity. In addition, since the hybrid multi-core architecture allows different cores to have varying ISAs, designers will have the flexibility to select appropriate ISAs tailored for specific application domains, which is particularly important for embedded applications with power, memory size and cost constraints.

Researchers have also explored implicit (speculative) multithreaded processors, such as multiscalar [35] and trace processor [36] as well as distributed uniprocessors, such as Grid [37] and RAW processors [38]. While these architectures can also improve serial performance either by hardware or compiler-based parallelization, they require fundamentally new hardware design and execution model. In contrast, the hybrid multi-core can reuse VLIW and superscalar cores to mitigate hardware complexity.

In the domain of embedded systems, however, there are some heterogeneous dual-core processors typically consisting of a RISC processor and a DSP, such as TI OMAP5910 [44] or Philips Nexperia [44]. In these processors, the RISC processor is used to orchestra commands and control, while the DSP is used for computation-intensive signal processing tasks. Both these two processors [43, 44], however, are designed for specialized domain, i.e., the mobile multimedia applications. By comparison, this paper proposes a VLIW/superscalar hybrid multi-core architecture for general-purpose applications.

Recently, IBM introduced the Cell multiprocessor [49]. The first-generation Cell processor combines a 64-bit Power-Architecture-compliant Power processor element (PPE) with eight newly architected synergistic processor elements (SPEs) through a coherent on-chip element interconnect bus (EIB). Conceptually, the Cell processor can be considered as a specific type of hybrid multi-core processors, since the PPE and SPEs have different instruction sets. However, the Cell SPEs are geared towards single-precision SIMD computation, and thus rely on the general-purpose PPE to perform all the normal operations. In contrast, the hybrid multi-core architecture consists of general-purpose VLIW and superscalar processors, and all the cores on the hybrid multiprocessor can be either used cooperatively for reducing single-threaded latency or used independently and concurrently to support general-purpose application for higher throughput. Also, the main objective of the Cell processor is to provide outstanding performance on game/multimedia applications; while the hybrid multi-core architecture is a more broad architecture aiming at achieving both high throughput and low latency for general-purpose applications.

7. Concluding Remarks and Future Work

This paper presents the VLIW/superscalar heterogeneous multi-core architecture, which can combine the advantages of both VLIW and superscalar processors to improve single-threaded performance while keeping the flexibility to benefit multithreaded applications. The proposed architecture is friendly to compilers, which can potentially shift part of the hardware complexity to the compiler to better exploit thread-level parallelism, either non-speculative or speculative, as well as instruction-level parallelism for improving single-threaded performance. In contrast, the single-ISA heterogeneous multi-core architecture [1] generally relies on a complex superscalar core to improve single-threaded performance, where compiler can only play a limited role while the hardware complexity is overwhelming with diminishing return. In addition to the prospective performance benefits, the proposed hybrid architecture can also be potentially more energy-efficient if a large fraction of computation can be performed by the VLIW core, since the work done by the compiler does not need to be repeated by the VLIW hardware (as compared to pure superscalar cores).

In this paper, our initial quantitative evaluation of this hybrid multi-core architecture focuses on investigating the performance potential of a hybrid dual-core processor by running the pre-execution thread on the superscalar core to mitigate the L2 stalls for the VLIW core through the shared L2 cache. Our experimental results demonstrate that for applications suffering from severe L2 cache misses, pre-execution by the superscalar core is very effective at reducing the L2 cache stalls of the main thread on the VLIW core, leading to the improvement of single-threaded performance by up to 40.8%. These preliminary results also indicate that heterogeneous processors with different architectures can work cooperatively within a multi-core architecture to boost single-threaded performance effectively.

Currently, we are implementing the thread partitioning phase of the hybrid compiler to better harness the potential of the hybrid multi-core processors. In our future work, we plan to systematically explore the architectural design space of the VLIW core(s), the superscalar core(s) and their interconnections for making better tradeoffs for the hybrid multi-core architecture. While this paper concentrates on studying a VLIW/superscalar dual-core, we also intend to investigate the scalability of hybrid multi-cores with different number of VLIW and superscalar processors. Moreover, besides the serial and parallel performance, we will consider other important design constraints, such as power, temperature, area, to find the optimal design points for this hybrid multi-core architecture.

Acknowledgement

This work was funded in part by NSF grant 0613244.

8. References

[1] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In Proc. of the 31th Annual International Symposium on Computer Architecture (ISCA'04), June 2004.

[2] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In Proc. of the 38th IEEE/ACM International Symposium on Microarchitecture, November 2005

[3] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson and K. Chang. The case for a single-chip multiprocessor. In Proc. of the Seventh International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), October 1996.

[4] S. Sudharsanan, P. Sriram, H. Frederickson and A. Gulati. Image and video processing using MAJC 5200. In Proc. of ICIP 2000.

[5] U. Banerjee. Loop parallelization. Kluwer Academic Publishers, Boston, MA, 1994.

[6] M. K. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In Proc. of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2003.

[7] J. T. Oplinger, D. L. Heine and M. S. Lam. In search of speculative thread-level parallelism. In Proc. of the International Conference on Parallel Architectures and Compilation Techniques, October 1999.

[8] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In Proc. of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, 2004.

[9] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In the 4th International Symposium on High-Performance Computer Architecture, Feb 1998.

[10] J. Y. Tsai, J. Huang, C. Amlo, D. J. Lilja and P.-C. Yew. The superthreaded processor architecture. IEEE Transactions on Computers, 1999.

[11] M. Schlansker and B. R. Rau. EPIC: An architecture for instruction-level parallel processors. HP Labs Technical Reports, 1999.

[12] N. Vachharajani, M. Iyer, C. Ashok, M. Vachharajani, D. I. August, and D. A. Connors. Chip multi-processor scalability for single-threaded applications. In Proc. of the 2005 Workshop on Design, Architecture and Simulation of Chip Multi-Processors (dasCMP), November 2005.

[13] S. S. Muchnick. Advanced compiler design and implementation. Morgan Kaufmann Publishers, 1997.

[14] T. Ball and J. R. Larus. Branch prediction for free. In Proceedings of SIGPLAN Conference on Programming Language Design and Implementation, 1993.

[15] S. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of load/store instruction latencies. In Proc. of the 26th Annual International Symposium on Microarchitecture, December 1993.

[16] R. Hank, W. Hwu, and B. Rau. Region-based compilation: an introduction and motivation. In Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture, pages 158-168, Ann Arbor, MI, Nov, 1995.

[17] S. Liao, P. Wang, H. Wang, G. Hoflehner, D. lavery, and J. Shen. Post-pass binary adaptation for software-based speculative precomputation. In Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2002.

[18] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In Proc. of the 28th International Symposium on Computer Architecture, June 2001.

[19] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. lavery and J. Shen. Speculative precomputation: long range prefetching of delinquent loads. In Proc. of the 28th Annual International Symposium on Computer Architecture, June 2001.

[20] J. Brown, H. Wang, G. Chrysos, P. Wang and J. Shen. Speculative precomputation on chip multiprocessors. In Proc. of the 6th Workshop on Multithreaded Execution, Architecture and Compilation, Nov 2002.

[21] Y. Song, S. Kalogeropulos and P. Tirumalai. Design and implementation of a compiler framework for helper threading on multi-core processors. In Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques, Sep 2005.

[22] V. Kathail, M. S. Schlansker and B. R. Rau. HPL-PD architecture specification: version 1.1. HPL Technical Report, 2000.

[23] L. Gwennap. Digital 21264 sets new standard. Microprocessor Report, Oct. 28, 1996.

[24] Crusoe processor, model TM5800. Transmeta Corp. White Paper, 2001.

[25] J. A. Fisher, P. Faraboschi and C. Young. Embedded computing: a VLIW approach to architecture, compilers, and tools. Morgan Kaufmann Publishers, 2005.

[26] Trimaran homepage, http://www.trimaran.org.

[27] D. C. Burger and T. M. Austin. The SimpleScalar tool-set, Version 2.0. Technical Report 1342, Dept. of Computer Science, UW, June, 1997.

[28] SPEC homepage, http://www.spec.org.

[29] E. Altman, M. Gschwind, S. Sathaye, S. Kosonocky, A. Bright, J. Fritts, P. Ledak, D. Appenzeller, C. Agricola and Z. Filan. BOA: the architecture of a binary translation processor. IBM Research Report, 1999.

[30] S. P. Vanderwiel, D. J. Lilja. Data prefetch mechanisms. ACM Computing Surveys, June, 2000.

[31] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horwitz. Architectural support for copy and tamper resistant software. In Proc. of the ACM 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Nov. 2000.

[32] T. Kgil, L. Falk and T. Mudge. ChipLock: support for secure microarchitecture. In Proc. of the ACM SIGARCH Computer Architecture News, Vol. 33, No. 1, March 2005.

[33] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Water, and W. Hwu. IMPACT: an architectural framework for multiple-instruction-issue processors. In Proc. of the 18th Annual International Symposium on Computer Architecture, May, 1991.

[34] J. N. Amaral, G. Gao, E. D. Kocalar, P. O'Neill, and X. Tang. Design and implementation of an efficient thread partitioning algorithm. In Proc. of the 36th International Symposium on High Performance Computing, 2000.

[35] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In Proc. of the 22th International Symposium on Computer Architecture, 1995.

[36] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In Proc. of Micro 30, 1997.

[37] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of grid processor architectures. In Proc. of the International Symposium on Microarchitecture, 2001.

[38] M. B. Taylor et al. The raw microprocessor: a computational fabric for software circuits and general purpose programs. In Proc. of IEEE Micro, Mar/Apr 2002.

[39] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In Proc. of the 27th Annual International Symposium on Microarchitecture, 1994.

[40] J. A. Fischer. Very long instruction word architectures and the ELI-512. In Proc. of the 10th Symposium on Computer Architectures, pp. 140-150, IEEE, June, 1983.

[41] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. The Journal of Supercomputing, 1993.

[42] S. A. Mahilke, D. C. Lin, W. Y. Chen, R. E. Hank and R. A. Bringmann. Effective compiler support for predicated execution using hyperblock. In Proc. of the 25th International Symposium on Microarchitecture, pp.45-54, Dec. 1992

[43] OMAP5910 Dual-Core Processor Datasheet. Texas Instruments White Paper, Aug 2004.

[44] Nexperia PNX4103 mobile multimedia processor. Philips White Paper, Jan 2006.

[45] B. R. Rau. Dynamically scheduled VLIW processors. In Proc. of 26th Annual International Symposium on Microarchitecture, Dec. 1993.

[46] T. M. Conte, S. W. Sathaye. Properties of rescheduling size invariance for dynamic rescheduling-based VLIW cross-generation compatibility. IEEE Transactions on Computers, Vol 49, Issue 8, August 2000.

[47] R. Ju, S. Chan, C. Wu, R. Lian and T. Tuo. Open research compiler (ORC) for Itanium processor family. MICRO-34 Tutorial, Dec. 2001.

[48] S. Balakrishnan, R. Rajwar, M. Upton and K. Lai. The impact of performance asymmetry in emerging multi-core architectures. In Proc. of the 32th annual International Symposium on Computer Architecture, June 2005

[49] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer and D. Shippy. Introduction to the Cell multiprocessor. In IBM Journal of Research and Development, Vol. 49, No. 4/5, July/September, 2005.