Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications

Hongtao Zhong, Steven A. Lieberman, and Scott A. Mahlke
Advanced Computer Architecture Laboratory
University of Michigan, Ann Arbor, MI 48109
{hongtaoz,lieberm,mahlke}@umich.edu

Abstract

Chip multiprocessors with multiple simpler cores are gaining popularity because they have the potential to drive future performance gains without exacerbating the problems of power dissipation and complexity. Current chip multiprocessors increase throughput by utilizing multiple cores to perform computation in parallel. These designs provide real benefits for server-class applications that are explicitly multi-threaded. However, for desktop and other systems where single-thread applications dominate, multicore systems have yet to offer much benefit. Chip multiprocessors are most efficient at executing coarse-grain threads that have little communication. However, general-purpose applications do not provide many opportunities for identifying such threads, due to frequent use of pointers, recursive data structures, if-then-else branches, small function bodies, and loops with small trip counts. To attack this mismatch, this paper proposes a multicore architecture, referred to as Voltron*, that extends traditional multicore systems in two ways. First, it provides a dual-mode scalar operand network to enable efficient inter-core communication and lightweight synchronization. Second, Voltron can organize the cores for execution in either coupled or decoupled mode. In coupled mode, the cores execute multiple instruction streams in lock-step to collectively function as a wide-issue VLIW. In decoupled mode, the cores execute a set of fine-grain communicating threads extracted by the compiler. This paper describes the Voltron architecture and associated compiler support for orchestrating bi-modal execution.*

1 Introduction

Since the earliest processors came onto the market, the semiconductor industry has depended on Moore's Law to deliver consistent application performance gains through the multiplicative effects of increased transistor counts and higher clock frequencies. However, power dissipation and thermal issues have emerged as dominant design constraints that severely restrict the ability to increase clock frequency to improve performance. Exponential growth in transistor counts still remains intact and a powerful tool to improve performance. This trend has led major microprocessor companies to put multiple processors onto a single chip. These multicore systems increase throughput and efficiency by utilizing multiple simpler cores to perform computation in parallel and complete a larger volume of work in a shorter period of time. Such designs are ideal for servers where coarse thread-level parallelism (TLP) is abundant. But for systems where single-threaded applications dominate, multicore systems offer limited benefits.

One of the critical challenges going forward is whether the available hardware resources in multicore systems can be converted into meaningful single-thread application performance gains. In the scientific domain, there is a long history of successful automatic parallelization efforts [11, 9]. These techniques target counted loops that manipulate array accesses with affine indices, where memory dependence analysis can be precisely performed. Loop- and data-level parallelism are extracted to execute multiple loop iterations or process multiple data items in parallel. Unfortunately, these techniques do not translate well to general-purpose applications.

Extracting TLP from general-purpose applications is difficult for a variety of reasons that span both the design limitations of current multicore systems and the characteristics of the applications themselves. On the hardware side, processors in multicore systems communicate through memory, resulting in long latency and limited bandwidth communication. Synchronization of the processors is also performed through memory, thereby causing a high overhead for synchronization. Finally, multicore systems do not support exploiting instruction-level parallelism (ILP) across multiple cores in an efficient manner. In short, current multicores are generally direct extensions of shared-memory multiprocessors that are designed to efficiently execute coarse-grain threads. On the application side, the abundance of dependences, including data, memory, and control dependences, severely restrict the amount of parallelism that can be extracted from general-purpose programs. The frequent use of pointers and linked data structures is a particularly difficult problem to overcome even with sophisticated memory alias analysis [18]. Coarse-grain parallelism often cannot be discovered from general-purpose applications. Frequent communication and synchronization along with modest amounts of parallelism are dominant characteristics. Thus, there is a distinct mismatch between multicore hardware and general-purpose software.

To effectively deal with mapping general-purpose applications onto multicore systems, new architectural support is needed that is capable of efficiently exploiting the diversity of parallelism within these applications. These *hybrid* forms of parallelism include ILP, memory parallelism, loop-level parallelism, and fine-grain TLP. To accomplish this goal, this paper proposes a multicore architecture, referred to as *Voltron*. Voltron extends conventional multicore systems by providing architectural support to configure the organization of the resources to best exploit the hybrid forms of parallelism that are present in an application. There are two primary operation modes: coupled and decoupled. In coupled mode, each core operates in lock-step with all other cores forming a wide-issue VLIW processor. Through com-

1-4244-0805-9/07/$25.00 ©2007 IEEE

piler orchestrated control flow, coupled mode exploits ILP across multiple instruction streams that collectively function as a single-threaded stream executed by a VLIW processor. Although cores fetch independently, they execute instructions in the same logical location each cycle, and branch to the same logical target. In decoupled mode, all cores operate independently on separate fine-grain threads. The compiler slices an application into multiple, communicating subgraphs that are initiated using a lightweight thread spawn mechanism operating in the same program context. Decoupled mode offers the opportunity to efficiently exploit TLP and memory parallelism using fine-grain, communicating threads.

There are several key architectural features of Voltron that enable efficient execution and flexible configurability. First, a lightweight operand network provides fast inter-processor communication in both execution modes. The operand network provides a direct path for cores to communicate register values without using the memory. Second, a compiler-orchestrated distributed branch architecture enables multiple cores to execute multi-basic block code regions in a completely decentralized manner. This enables single-threaded, branch-intensive applications to be executed without excessive thread spawning and synchronization barriers. Finally, flexible memory synchronization support is provided for efficient handling of explicit memory communication and ambiguous memory dependences.

A obvious alternative for exploiting hybrid forms of parallelism is a heterogeneous multicore system. In a heterogeneous design [12], one or more powerful cores execute single thread applications to exploit ILP, while many simpler cores execute explicitly threaded applications. In this manner, different portions of the multicore system are designed to exploit the best form of available parallelism. The major problem of this approach is the manufacturing difficulty. The more powerful core usually runs at a higher frequency and has a different supply voltage. It is hard to fabricate cores with different process requirements on the same die. Further, wide-issue cores for ILP introduce design cost and complexity into the overall system. On the other hand, Voltron utilizes multiple homogeneous simpler cores whose execution can be adapted to the available application parallelism for both single and multi-thread applications. This approach reduces system complexity and increases hardware utilization. Note that a heterogeneous multicore could also include special purpose cores to accelerate multimedia, networking, or encryption computations. Such accelerators provide good efficiency for the supported types of computation. We consider accelerators an orthogonal issue as they offer similar benefits to any multicore system.

The Voltron architecture is inspired by prior work on multicluster VLIW processors [3, 27, 21, 28] and the RAW architecture [24]. This work shares the central concept of software-controlled distributed ILP processing in prior works. However, it provides the flexibility to configure the processor resources in multiple modes to best extract the available parallelism. Further, it provides new architectural mechanisms for lightweight communication and synchronization. This paper provides a description of the Voltron architecture and the associated compiler support.

2 Parallelization Opportunities in Single-thread Applications

The challenge with increasing single-thread application performance on multicore systems is identifying useful parallelism that can be exploited across the cores. We examined a set of applications from the MediaBench and SPEC suites to identify forms of parallelism that are feasible to automatically extract by a compiler. The parallelism consists of three types: instruction level parallelism (ILP), fine-grain thread level parallelism (fine-grain TLP), and loop-level parallelism (LLP).

Instruction Level Parallelism. Modern superscalar and VLIW architectures successfully exploit ILP in single-thread applications to provide performance improvement. Most applications exhibit some ILP, but it is often highly variable even within an application. We observed program regions with high degrees of ILP (6-8 instructions per cycle), followed by mostly sequential regions. Overall ILP results can also be low due to frequent memory stalls.

Exploiting ILP in a multicore system requires low latency communication of operands between instructions so the dependences in the program can be satisfied quickly. Most architectures pass operands through shared register file and bypass networks, which are absent between cores in multicore systems. Multicluster VLIW [3, 27, 21, 28] exploits ILP across multiple clusters with separate register files. Networks connecting the clusters transmit operands between registers with very low latency. Mechanisms similar to multicluster VLIW must be provided in to efficiently exploit ILP in multicore architectures.

Fine-Grain Thread Level Parallelism. Parallelization opportunities can also be created by slicing program regions into multiple communicating sequential subgraphs or fine-grain threads. Fine-grain TLP allows concurrent execution of instructions as well as the overlapping of memory latencies. The memory-level parallelism (MLP) achieved by overlapping cache misses or misses with computation has large potential to increase performance. Conventional superscalars must create opportunities for such parallelism with large instruction windows. But, the compiler can expose opportunities across much larger scopes and with much simpler hardware.

We investigated two forms of fine-grain TLP: decoupled software pipelining (DSWP) and strand decomposition. Recent research on DSWP [19] exploits fine-grain TLP in loop bodies. The execution of a single iteration of a loop is subdivided and spread across multiple cores. When the compiler can create subdivisions that form an acyclic dependence graph, each subpart can be independently executed forming a pipeline. DSWP allows better utilization of cores and better latency tolerance when such pipeline parallelism can be extracted from the program.

Strand decomposition refers to slicing program regions into a set of communicating subgraphs. These subgraphs are overlapped to exploit ILP and MLP. A region decomposed into strands is illustrated in Figure 1. Nodes in the graph are instructions and edges represent register flow dependences. The gray nodes represent load instructions. The dotted line divides the instructions into two fine-grain threads. The compiler inserts communication/synchronization instructions to transfer data between threads for all inter-thread dataflow. One of the critical sources of speedup in the strands is MLP. Strands must be carefully identified to allow overlap of memory instructions and any cache misses that result.

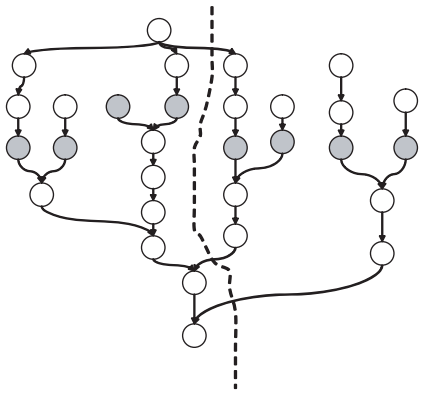

Figure 1: Example of fine-grain TLP.

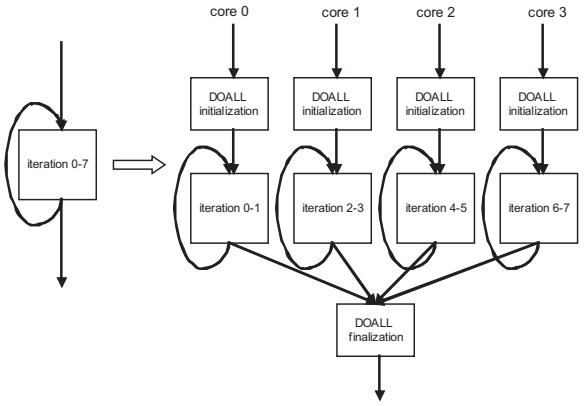

Figure 2: Example of loop-level parallelism.

Low latency asynchronous communication between cores is required to exploit fine-grain ILP on multicore architectures. High communication latency between cores can easily outweigh the benefit of fine-grain TLP. Traditional multicore systems do not support fine-grain ILP because the only way to communication between cores is through the memory. One of the main focuses of this paper is to enable execution of such threads with a low latency communication network.

Loop-level Parallelism. Exploiting loop-level parallelism has generally been utilized in the scientific computing domain. Most commonly, DOALL loops are identified where the loop iterations are completely independent from one another. Thus, they can execute in parallel without any value communication or synchronization. When enough iterations are present, DOALL loops can provide substantial speedup on multicore systems. Figure 2 illustrates the execution of a DOALL loop on a 4-core system. Automatic parallelization of DOALL loops has been widely studied [11, 9] and shown a large degree of success on a variety of numerical and scientific applications.

With general-purpose applications, the consensus is that DOALL parallelism does not exist to any significant level. Due to extensive use of pointers, complex control flow, and recursive data structures, it is extremely difficult for compilers to identify parallel loops. However, we found that a fair amount of statistical DOALL parallelism [14] does exist in general-purpose applications. Statistical DOALL loops are loops that do not show any cross-iteration dependences during profiling, although the independence cannot be proven by the compiler. Such loops can be speculatively parallelized with some lightweight hardware support to detect

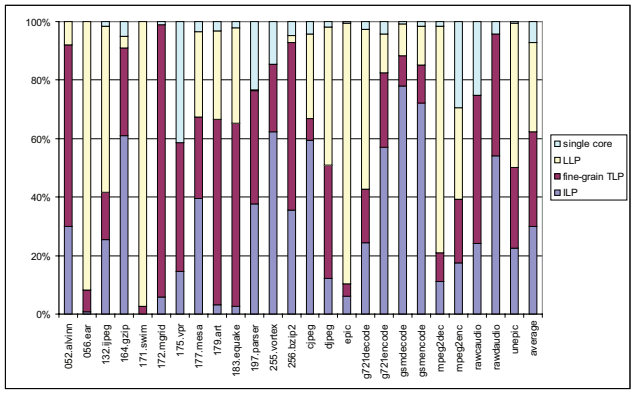

Figure 3: Breakdown of exploitable parallelism for a 4-core system.

mis-speculation and rollback the execution if needed. Besides, the loops in general purpose applications tend to be smaller and have fewer iterations than scientific applications. Thus, the overhead of spawning threads must be small to make this parallelism profitable to exploit. Further, DOALL parallelism is often hidden beneath the surface. The compiler must utilize sophisticated pointer analysis to understand the memory reference behavior [18] and new optimizations are needed to untangle register dependences due to scalars, such as induction or accumulator variables.

Parallelism Breakdown. To understand the availability of each of the three types of parallelism, we conducted an experiment to classify the form of parallelism that was best at accelerating individual program regions for a set of single-thread applications from the SPEC and MediaBench suites. The experiment assumes a 4-core system where each core is a single-issue processor. Further, the system is assumed to contain mechanisms for fast inter-core communication and synchronization as discussed in Section 3. Figure 3 shows the fraction of dynamic execution that is best accelerated by exploiting each type of parallelism. The benchmarks were compiled to exploit each form of parallelism by itself. On a region-by-region basis, we choose the method that achieves the best performance and attribute the fraction of dynamic execution covered by the region to that type of parallelism.

Figure 3 shows there is no dominant type of parallelism, and the contribution of each form of parallelism varies widely across the benchmarks. On average, the fraction of dynamic execution that is attributed to each form of parallelism is: 30% by ILP, 32% by fine-grain TLP (12% percent by DSWP and 20% by strands), and 31% by LLP. 7% of the execution doesn't show any opportunities for all three types of parallelism as it had the highest performance on a single core.

The results show that no single type of parallelism is a silver bullet in general-purpose applications. The types of parallelization opportunities vary widely across the applications as well as across different regions of an application. To successfully map these applications onto multicore systems, the architecture must be capable of exploiting hybrid forms of parallelism, and further the system must be adaptable on a region-by-region basis. This finding is the direct motivation for the Voltron architectural extensions that are presented in the next section.

3 Voltron Architecture

To efficiently exploit the different types of parallelism available in single-thread applications, including ILP, fine-grain TLP, and LLP, Voltron extends a conventional multicore architecture with a scalar operand network and supports two execution modes. The scalar operand network supports low latency communication of operands between register files of the cores to enable parallel execution and synchronization across the cores. The Voltron processor can operate in two different modes: coupled and decoupled. Having these two modes provides a trade-off between communication latency and flexibility. Depending on the type of parallelism being exploited, Voltron can execute in the coupled mode and behave like a multicluster VLIW to exploit ILP, or execute in the decoupled mode to exploit fine-grain TLP and LLP.

Figure 4(a) shows an overall diagram of a four-core Voltron system. The four cores are organized in a two-dimensional mesh. Each core is a single-cluster VLIW processor with extensions to communicate with neighboring cores. Cores have private L1 instruction and coherent data caches, and all four cores share a banked L2 cache and main memory. The cores access a unified memory space; the coherence of caches is handled by a bus-based snooping protocol. The Voltron architecture does not limit the way in which coherence between cores is maintained; any hardware or software coherence protocol will suffice. A scalar operand network connects the cores in a grid. The network includes two sets of wires between each pair of adjacent cores to allow simultaneous communication in both directions. The topology of the cores and the latency of the network is exposed to the compiler, which partitions the work of a single-threaded program across multiple cores and orchestrates the execution and communication of the cores. A 1-bit bus connecting all 4 cores propagates stall signals in the coupled execution mode (discussed in section 3.2).

Figure 4(b) shows the datapath architecture of each core, which is very similar to that of a conventional VLIW processor. Each core has a complete pipeline, including an L1 instruction and data cache, an instruction fetch and decode unit, register files, and function units (FUs). The execution order of instructions within a core is statically scheduled by the compiler. A Voltron core differs from a conventional VLIW processor in that, in addition to other normal functional units such as an integer ALU, a floating-point ALU, and memory units, each core has a communication unit (CU). The CU can communicate with other cores in the processor through the operand network by executing special communication instructions.

A low-cost transactional memory [7] is used to support parallel execution of statistical DOALL loops. The iterations of a DOALL loop are divided into several chunks, each forming a transaction. The transactions speculatively execute in parallel across multiple cores. The low cost transactional memory detects cross-core memory dependences and rolls back the memory state if memory dependence violation is detected. The compiler is responsible for roll back of the register state. More details on parallel execution of statistical DOALL loops are discussed in [14].

3.1 Dual-mode Scalar Operand Network

The dual-mode scalar operand network supports two ways to pass register values between cores to efficiently exploit different types of parallelism. Although both ILP and fine-grain TLP execution need low latency communication between cores, they have different requirements for the network. ILP execution performance is very sensitive to the communication latency, thus it is important to make the latency as low as possible for ILP execution. On the other hand, the execution of multiple fine-grain threads are decoupled, thus fine-grain threads are less sensitive to the latency. However, fine-grain threads require asynchronous communication between cores; therefore, queue structures must be used to buffer values. The Voltron scalar operand network supports two modes to meet the requirement of ILP and fine-grain TLP execution: a direct mode that has a very low latency (1 cycle per hop) but requires both parties of the communication to be synchronized, and a queue mode that has a higher latency (2 cycles + 1 cycle per hop) but allows asynchronous communication.

The scalar operand network in Voltron consists of communication components in every core and links between cores. Figure 4(c) shows the detail of the CU in a Voltron core. A core performs communication with another core through the scalar operand network by issuing special instructions to the communication unit. The operation of the CU depends on the communication mode.

Direct mode. Direct mode communication allows two adjacent cores to communicate a register value in one cycle. Two new operations are added to the instruction set architecture (ISA) to support direct mode communication: PUT and GET. The PUT operation has two operands, a source register identifier and a 2-bit direction specifier. A PUT reads the value from source register and puts the value to the bus specified by the direction specifier (east, west, north or south). The GET operation also has two operands, a destination register id and a direction specifier. GET gets a value from the bus specified by the direction specifier and directly writes it into the destination register. A pair of adjacent cores communicate a register value by executing a PUT operation and a GET operation on two cores at the same cycle. Jointly executed PUT/GET operations function collectively as an inter-cluster move in a traditional multicluster VLIW [3].

The direct mode bypass wires in Figure 4(c) allow the communication unit to access the inter-core connections directly. Thus, the communication latency between two adjacent cores under direct mode is very low, only requiring one cycle to communicate a register value between neighboring cores. This mode of communication requires the cores to execute in lock-step (discussed in the next section) to guarantee PUT/GET operations are simultaneously executed. If a core needs to communicate with a non-adjacent core, a sequence of PUT and GET pairs are inserted by the compiler to move the data to the final destination through multiple hops.

Queue mode. Queue mode allows decoupled communication between cores. Under this mode, the core that produces a register value does not need to remain synchronized with the core that consumes the value. The send queue, receive queue, and the routing logic in Figure 4(c) are used to support this mode of communication. Two operations are added to the ISA to support queue mode communication: SEND and RECV. A SEND operation takes two operands, a source register number and a target core identifier. When a SEND operation is executed, it reads the value in the source register and writes a message to the send queue in that core. The message contains the value from the sender core as well as the target core identifier. A RECV operation analogously takes two operands, a destination register number

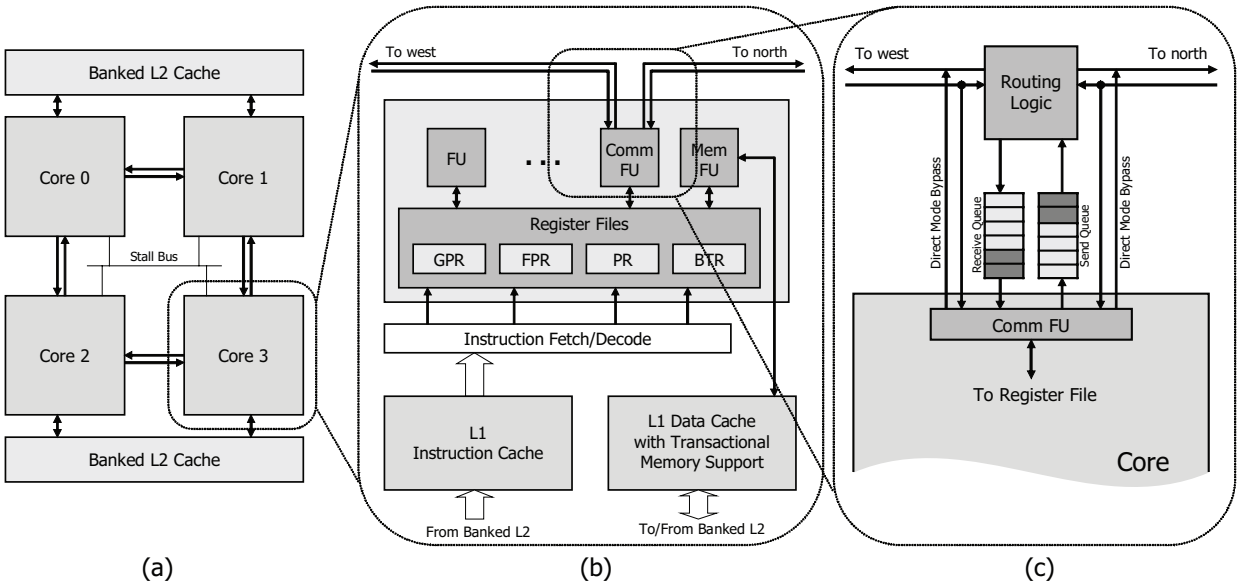

Figure 4: Block diagram of the Voltron architecture: (a) 4-core system connected in a mesh topology, (b) Datapath for a single core, and (c) Details of the inter-core communication unit.

and a sender core identifier. When a RECV operation is executed, it looks for the message from the specified sender in the receive queue, and writes the data to the destination register if such a message is found. It stalls the core if such a message is not found. The receive queue uses a CAM structure to support fast sender identifier lookup.

The router gets items from the send queue and routes them to the target core through one or more hops. In queue mode, SEND and RECV operations do not need to be issued in the same cycle. Data will wait in the receive queue until the RECV is executed, and the receiver core will stall when a RECV is executed and the data is not available. Only one pair of SEND/RECV operations is required to communicate between any pair of cores; the router will find a path from the sender to the receiver. The latency of communication between cores in the queued mode is $2+number\ of\ hops$: it takes one cycle to write the value to the send queue, one cycle per hop to move the data, and one cycle to read data from the receive queue. The operand network operating in queue mode is similar to the RAW scalar operand network [25].

The two modes of communication provide a latency/flexibility trade-off. The compiler can examine the characteristics of an application to utilize direct mode when communication latency is critical and queue mode when non-deterministic latencies caused by frequent cache misses dominate.

3.2 Voltron Execution Modes

Voltron supports two execution modes that are customized for the form of parallelism that is being exploited: coupled and decoupled. Coupled efficiently exploits ILP using the direct mode operand network, while decoupled exploits LLP and fine-grain TLP using the queue mode operand network.

Coupled mode. In coupled mode, all cores execute in lock-step, collectively behaving like a wide-issue multicluster VLIW machine. The cores pass register values to each other using the direct communication mode. The compiler is responsible for partitioning computation, scheduling the

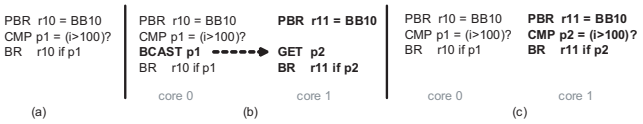

Figure 5: Execution of a distributed branch: (a) Original unbundled branch, (b) Branch with predicate broadcast, and (c) Branch with the condition computation replicated.

instructions, and orchestrating the communication between cores. In this mode, Voltron operates as a distributed VLIW (DVLIW) machine, which differs from conventional multicluster VLIW in that there is no centralized fetch unit; each core maintains its own control flow [28]. The branch mechanism in Voltron is based on the unbundled branch in the HPL-PD ISA [8]. In HPL-PD, the portions of each branch are specified separately: a prepare-to-branch (PBR) operation specifies the branch target address, a comparison (CMP) operation computes the branch condition, and a branch (BR) transfers the control flow of the program based on the target address and the branch condition, as illustrated in Figure 5(a).

Figure 5(b) illustrates the branch mechanism in Voltron coupled mode. To synchronize the control flow in all cores, each core specifies its own branch target using separate PBR operations. The branch target represents the same logical basic block, but a different physical block as the instructions for each core are located in different memory spaces. The branch condition is computed in one core and broadcast to all the other cores using a BCAST operation. Other cores receive the branch condition using a GET operation to determine if the branch is taken or fall through. (Note, the branch condition can alternatively be replicated on all cores as shown in Figure 5(c)). BR operations are replicated across all cores and scheduled to execute in the same cycle. When the BR operations are executed, every core branches to its own branch target (same logical target) keeping all cores synchronized. In essence, separate instruction streams are executed on each

core, but these streams collectively function as a single logical stream on a conventional VLIW processor.

If one core stalls due to cache misses, all the cores must stall to keep synchronized. The 1-bit stall bus is used to propagate this stall signal to all 4 cores. For a multicore system with more than 4 cores, propagating the stall signal to all cores within one cycle may not be realistic. We solve this problem using the observation that coupling more than 4 cores is rare as it only makes sense when there are extremely high levels of ILP. Therefore, cores are broken down into groups of 4 and coupled mode execution is restricted to each group. Of course, decoupled mode can happen within the groups or across groups when exploiting LLP and fine-grain TLP.

To illustrate coupled mode execution, Figure 6(a) shows the control flow graph (CFG) of an abstract code segment. Each node in the graph represents a basic block in the single-thread program. Figure 6(b) illustrates the code segment executing in coupled mode on a two-core system. The compiler partitions each block into two blocks, one for each core, using a multicluster partitioning algorithm [4]. For example, the operations in block A are divided into A.0 and A.1 to execute on core0 and core1, respectively. The schedule lengths of any given block are the same across all the cores; if they differ, the compiler inserts NO_OPs to ensure they match. All cores execute in lock-step and communicate through PUT and GET operations (not shown in the figure). When core0 stalls due to a cache miss in A.0, both cores have to stall to keep the execution synchronized. Similarly, when core1 stalls in D.1, both cores stall. Every branch in the original code is replicated to all cores. For conditional branches, such as the branch at the end of block A, the branch condition is computed in one core and broadcast to the others.

The most important benefit of coupled mode is the low communication latency of using the operand network in direct mode. Coupled mode can handle frequent inter-core communication without increasing the schedule length by a large amount. The drawback is its lockstep execution. If one core stalls, all cores stall. The stalling core prevents other cores from making any progress even if parallelism exists. Coupled execution mode is ideal for code with high ILP, predictable cache misses, and complicated data/memory dependences, which require a large amount of inter-core communication and can benefit from the low communication latency.

Decoupled Mode. The second mode of execution is decoupled where each core independently executes its own thread. As with a conventional multicore system, stalls on one core do not affect other cores. However, fast communication of register values through the scalar operand network allows Voltron to exploit a much finer grain of TLP than conventional multicore systems, which communicate through memory. The compiler automatically extracts fine-grain threads from a single-thread program using mechanisms described in Section 4. Multiple threads execute independently on multiple cores and communicate through the scalar operand network. The network operates in queue mode for decoupled execution.

A core initiates the execution of a fine-grain thread on another core by executing a SPAWN operation, which sends the starting instruction address of the fine-grain thread to be executed to the target core. The target core listens to the receive queue when it is idle. Upon the arrival of the instruction address, it moves the value to its program counter (PC) and starts executing from that address. All fine-grain threads share the same memory space and stack frame, thus there is no setup for a separate context required. The live-in values for the fine-grain thread are received through the operand network during execution or are loaded from memory. When the execution of a fine-grain thread finishes, it executes a SLEEP operation, and the core listens to the queue for the starting address of the next fine-grain thread. The register values in a core remain unchanged after a fine-grain thread terminates. The next fine-grain thread to execute on that core can use the values in the register file as live-in values.

Multiple fine-grain threads can collectively and efficiently execute the same multi-block region of the original single-thread program. In this case, the branch operations in the original code are replicated in all the fine-grain threads. The branch condition for each branch can be computed in one thread and broadcast to others, similar to the coupled mode. The differences here are that the branch conditions are broadcast using queue mode, and that the branches do not need to be scheduled in the same cycle. In decoupled mode, the computation of the branch conditions can also be replicated to other cores to save communication and reduce receive stalls. The compiler uses a heuristic to decide if the branch condition should be broadcast or computed locally for each branch.

The right hand side of Figure 6 illustrates the differences in partitioning and execution in decoupled mode. In decoupled mode, the compiler divides the application into fine-grain threads. Figure 6(c) depicts the threads extracted from the CFG shown in Figure 6(a). The first inner loop, which includes blocks A, B, C, and D, is partitioned between the cores. Block A is divided into two parts, A.0 and A.1 (similar to coupled mode), while all operations in block B are assigned to core0 and all operations in block C to core1. Block D is split analogously to block A.

Figure 6(d) shows the execution of these fine-grain threads in decoupled mode. Before core0 enters the first inner loop, a SPAWN operation is executed, which sends the start address of A.1 to core1. The execution of blocks A.0 and A.1 are independent; stalls that occur during A.0 do not affect A.1. The cores communicate register values using SEND and RECV operations (not shown in the figure). A.1 computes the branch condition for the block and broadcasts it to core0 using the scalar operand network in queue mode. After receiving the branch condition from core1, core0 branches to block B. Since the compiler assigned all operations in block B to core0, core1 directly jumps to block D. The branch as well as the branch condition computation is replicated across both cores in block D, thus broadcast and receive are not needed for this block. After two iterations, the execution exits the first inner loop, and the fine-grain thread in core1 finishes, executing a SLEEP operation. Block E in core0 will spawn another thread for the second inner loop to execute on core1. In this example, core0 behaves as the master, spawning jobs to core 1. This is the general strategy used by our compiler.

The most important benefit of decoupled mode is its tolerance of unexpected stalls, such as cache misses. Each core operates in its own stall domain and thus stalls independently. Memory parallelism can be exploited by overlapping cache misses on multiple cores or allowing other cores to make progress on their own thread as long as dependences are not violated. The fine-grain threads in Voltron provide a form of macro out-of-order execution across the cores, without the hardware and complexity of a wide-issue superscalar processor. Additionally, decoupled mode allows execution of traditional multithreaded programs.

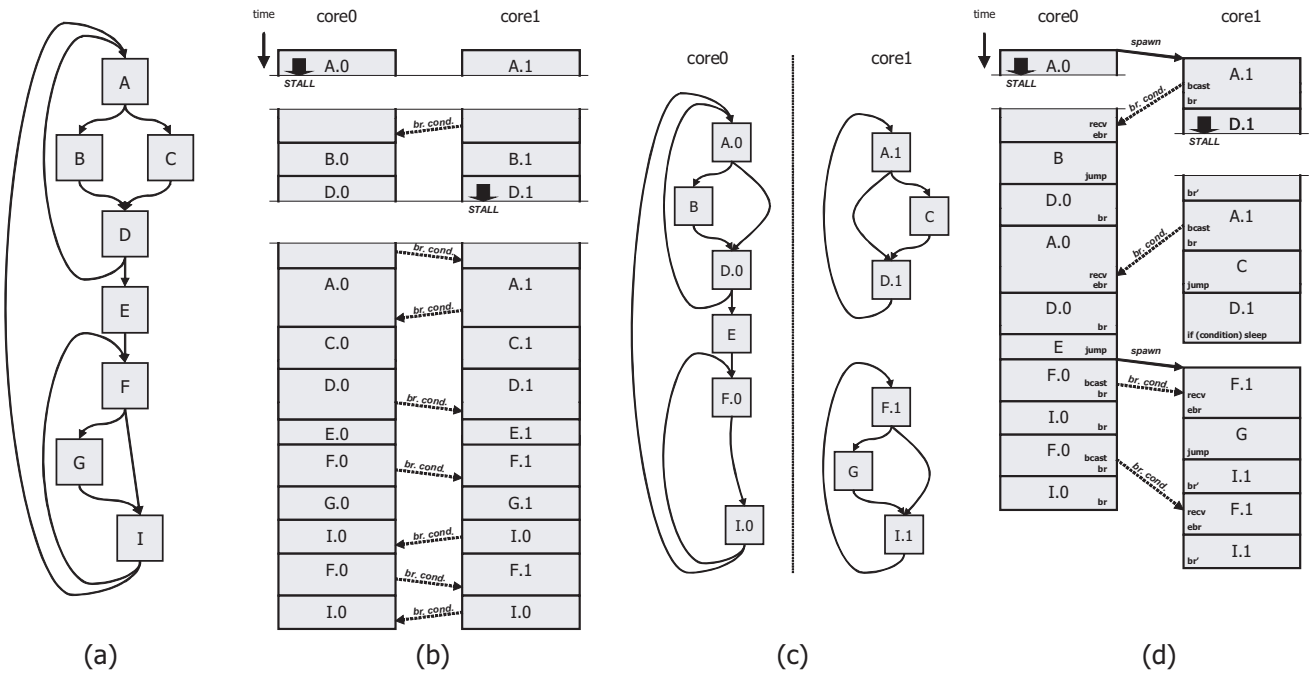

Figure 6: Code execution under different modes. (a) Control flow graph of a code segment. (b) Execution of the code segment under coupled mode. (c) Control flow graph of fine-grain threads extracted for decoupled mode. (d) Execution of the code segment under decoupled mode.

Switching Between Modes. Voltron supports fast mode switching by adding a new instruction MODE_SWITCH. MODE_SWITCH takes one literal operand specifying the new mode. When switching from coupled mode to decoupled mode, the compiler inserts MODE_SWITCH instructions in all cores and makes sure they are issued in the same cycle. The cores stop broadcasting stall signals and start using SEND/RECV instructions instead of PUT/GET to communicate register values after switching to decoupled mode. To switch from decoupled to coupled mode, the compiler inserts MODE_SWITCH instructions before the coupled code region in all cores. The MODE_SWITCH behaves as a barrier, so all cores wait until the last core reaches the switch to start synchronized execution. Note that the values in the register files are unchanged during the mode switch. This behavior facilitates running a single application in different modes and transparently switching between modes for different program regions.

3.3 Memory Dependence Synchronization

Thus far, the only communication that has been discussed has been the communication of register values. The cores can also communicate values through memory. Stores and loads that cannot be disambiguated by the compiler must execute in program order to ensure correct execution. In coupled mode execution, this simply requires the compiler to enforce both intra-core and inter-core memory dependences when the schedule is created. As long as dependent memory are executed in subsequent cycles, the hardware will ensure the correct data is obtained using the coherence mechanism.

In decoupled mode, the memory dependence problem is more difficult. We take the simple approach of just using the operand network to enforce the dependence. A SEND/RECV pair are used to send a dummy register value from the source of the dependence to the sink. Memory synchronization using SEND/RECV has two drawbacks. First, the SEND/RECV pair occupies the CUs and bandwidth without passing any useful value, which could delay other useful communication and consumes energy. Second, the SEND/RECV goes through message queues and routing logic, incurring a minimum latency of three cycles. Through our experimental evaluation, these drawbacks are not that serious. The compiler can perform sophisticated analysis to remove as many false dependences as possible [18]. Further, to reduce synchronization stalls, its important for the compiler to place dependent memory operations on the same core. Thus, in practice, the number of inter-core memory dependences is not large.

4 Compiling for Voltron

The execution of programs on Voltron is orchestrated by the compiler. We took advantage of advanced analysis, optimization, and transformation techniques and augmented them with our own novel algorithms to exploit different types of parallelism. We propose a strategy for classifying the types of parallelism available in each region, and deciding which to exploit. Section 4.1 describes several different compilation strategies, and Section 4.2 explains how the Voltron compiler selects which strategy to use for different regions of code (which ultimately determines the architectural execution mode). Note that compilation for multicore systems is a large research problem by itself and we can only provide an overview of the techniques that we utilize in the scope of this paper.

4.1 Compiler Techniques

Compiling for ILP. To exploit ILP, the compiler uses algorithms to partition applications for multi-cluster VLIW processors [6, 3]. Several effective partitioning algorithms have been proposed for multicluster VLIW [4, 2, 13, 17, 20]. For Voltron coupled mode, we employ the Bottom-Up Greedy (BUG) algorithm [4], which performs the partitioning before scheduling using heuristically-estimated scheduling times. The Voltron compiler performs BUG, determining an initial assignment of operations to cores, then selective replication of branches is performed to reduce inter-core communication. Finally, a second pass of BUG refines the partitions.

Extracting TLP with DSWP. We use the algorithm proposed in [19] to exploit pipeline parallelism. To perform DSWP partitioning, a dependence graph G, including data and control dependences, is created. The compiler detects strongly connected components (SCCs) in the dependence graph, and merges nodes in a SCC to a single node. As the SCCs include all the recurrences within the loop, the merged graph G_{scc} is acyclic. A greedy algorithm is employed to partition G_{scc}. Instructions are assigned to cores according to the partition.

Extracting strands using eBUG. The BUG partitioning algorithm tries to partition the operations to achieve minimum schedule length in all cores. When partitioning operations for fine-grain TLP, minimum schedule length does not necessarily guarantee minimum execution time. More factors need to be considered in the partitioning process. Through the analysis of programs and their execution on Voltron, we identified several factors that a partitioning algorithm must consider when identifying fine-grain strands:

- **Likely missing loads:** It is preferable to assign likely missing LOADs and the operations that depend on it to the same core. If the LOAD and its consumer are assigned to different cores, SEND/RECV operations must be inserted to move the value. When the LOAD misses in the cache, both the sender and the receiver must stall. This hurts performance as one of the cores could be doing useful work. Note that this issue does not occur in coupled mode because a load miss always stalls all cores.
- **Memory dependences:** If two memory operations can potentially access the same memory address, and one is a STORE, the second must be issued after the first. If they are assigned to different cores, the cores must be synchronized through SEND/RECV pairs. This synchronization has the potential to stall a core, so it is preferable to assign dependent memory operations to the same core.
- **Memory balancing:** It is beneficial to balance the data access on cores to effectively utilize the local caches even if the balancing causes increases to schedule length. Distributing independent memory accesses across cores also provides more opportunity to overlap stalls.

Taking these factors into account, we propose the Enhanced Bottom-Up Greedy (eBUG) partitioning algorithm for Voltron decoupled mode compilation. eBUG employs pointer analysis to identify dependent memory operations [18] and uses profiling information to identify loads likely to miss in the cache. eBUG first processes the dataflow graph, assigning a weight to each edge in the graph. Higher weights indicate edges that should not be broken during partitioning. High weight edges are assigned between LOADs that are likely to miss and the subsequent operations that consume the data. Memory dependence edges are also assigned higher weights to favor keeping dependent memory operations in the same strand.

After the edge weight assignment, eBUG performs the bottom-up assignment of operations. This part is similar to the original BUG algorithm [4]: the DFG is traversed in depth-first order, with operations on critical paths visited first. For each operation, the algorithm estimates the earliest possible completion time for each potential core assignment. The estimate considers the completion time of the operation's predecessors, possible communications to get operands from other cores, and possible communications to move results to the users. In eBUG, the weight on an edge is also added to the estimated completion time of the operation if the source and destination of the edge are assigned to different cores. eBUG also counts the number of memory operations already assigned to each core. When considering assignment of a memory operation to a particular core, a penalty is added to the estimated completion time if the core already has a majority of the memory operations assigned to it. This avoids capacity misses on heavily loaded cores and allows possible overlap of memory stalls and computation across different cores.

After partitioning is done, the compiler inserts SENDs and RECVs when a data flow edge goes across cores. Dummy SENDs and RECVs are also inserted to synchronize dependent memory operations on different cores. The compiler then replicates branches to construct the control flow for threads. Unlike the coupled mode, branches are only replicated to cores that contains control dependent instruction of the branch.

Extracting LLP from DOALL loops. The compiler performs memory profiling to identify loops that contain no profiled cross-iteration dependences, called statistical DOALL loops. Transformations, such as induction variable replication and accumulator expansion, are performed to eliminate false inter-iteration dependences. Once a statistical DOALL loop is identified, the compiler divides the loop iterations into multiple chunks, one for each core, to speculatively execute in parallel. The hardware monitors the coherence traffic between cores and roll back memory states if memory dependence violation is detected. The compiler is responsible for the rollback of register states in case of memory dependence violation [14].

4.2 Parallelism Selection

Parallelism can be extracted using the four methods described in the previous section. Many regions are eligible for multiple techniques, thus it is necessary to select a method that we expect to maximally expose parallelism. The selection strategy is relatively straight-forward.

Our strategy first looks for opportunities to parallelize loops with no profiled inter-iteration dependences, i.e., statistical DOALL loops. The compiler examines all loops in the program from the outermost nest to the innermost nest. If a loop is statistical DOALL and its trip count is greater than a threshold, LLP is extracted from the loop. DOALL loops are parallelized first because they provide the most efficient parallelism; neither communication nor synchronization is needed between cores for the parallel execution of the loop body.

```
for (i = 0; i < 8; ++i) {
  uf[i]  = u[i];
  rpf[i] = rp[i] * scalef;
}
```
(a) original C code

```
for (i1 = 0; i1 < 4; ++i1) {
  uf[i1]  = u[i1];
  rpf[i1] = rp[i1] * scalef;
}
```
(b) LLP code for core0

```
for (i2 = 4; i2 < 8; ++i2) {
  uf[i2]  = u[i2];
  rpf[i2] = rp[i2] * scalef;
}
```
(c) LLP code for core1

Figure 7: LLP from gsmdecode.

```
do {
} while (*(ush*)(scan+=2) == *(ush*)(match+=2) &&
         *(ush*)(scan+=2) == *(ush*)(match+=2) &&
         *(ush*)(scan+=2) == *(ush*)(match+=2) &&
         *(ush*)(scan+=2) == *(ush*)(match+=2) &&
         scan < strend);
```
(a) original C code

```
do {
  load r1 = *(ush*)(scan+=2);
  load r2 = *(ush*)(scan+=2);
  load r3 = *(ush*)(scan+=2);
  load r4 = *(ush*)(scan+=2);
  recv r5, core1;
  recv r6, core1;
  recv r7, core1;
  recv r8, core1;
  p1 = (r1==r5 && r2==r6 &&
    r3==r7 && r4==r8 &&
    scan < strend);
  send p1, core1;
} while (p1)
```
(b) fine-thread code for core0

```
do {
  load r11 = *(ush*)(match+=2);
  load r12 = *(ush*)(match+=2);
  load r13 = *(ush*)(match+=2);
  load r14 = *(ush*)(match+=2);
  send r11, core0;
  send r12, core0;
  send r13, core0;
  send r14, core0;
  recv p2, core0;
} while (p2)
```
(c) fine-thread code for core1

Figure 8: Strands from 164.gzip.

```
for (i = 8; i--;) {
  tmp1 = rrp[i];
  tmp2 = v[i];
  tmp2 =  ( tmp1 == MIN_WORD && tmp2 == MIN_WORD
          ? MAX_WORD
          : 0x0FFFF & (( (longword)tmp1 * (longword)tmp2
            + 16384) >> 15)) ;
  sri  = GSM_SUB( sri, tmp2 );
  tmp1  = ( tmp1 == MIN_WORD && sri == MIN_WORD
          ? MAX_WORD
          : 0x0FFFF & (( (longword)tmp1 * (longword)sri
           + 16384) >> 15)) ;
  v[i+1] = GSM_ADD( v[i], tmp1);
}
```

Figure 9: ILP from gsmdecode.

An example of such code is the loop shown in Figure 7 (a) from the gsmdecode benchmark. When the Voltron compiler encounters this loop, and it is compiling for a 2-core system, it divides it into a two equal chunks, shown in Figure 7 (b) and (c). We found this loop achieved a speedup of 1.9 over the serial code.

The compiler then looks for opportunities to exploit pipeline parallelism from loops using DSWP (i.e., unidirectional dependences). For all loops that haven't been parallelized by DOALL, the compiler uses DSWP to tentatively partition the flow graph. Speedup achieved from the partition is estimated; if the speedup is greater than a threshold (1.25 is used for our experiments), fine-grain threads are extracted from the partition. Otherwise the loop is left unchanged and handled using eBUG or BUG. For DSWP loops, the execution time after parallelization is limited by the execution time of the longest running partition. If a balanced DSWP partition is projected, we consider it superior to other fine-grain TLP and ILP techniques because the decoupled execution allows better overlapping of cache misses and communication latencies.

For the rest of blocks in the program, the compiler estimates cache miss stalls from profile information. If the time spent on cache misses is larger than a certain percentage of the estimated execution time of the block, fine-grain TLP will be exploited in decoupled mode because the decoupled execution can tolerate memory latencies better. The loop seen in Figure 8 (a) comes from the 164.gzip benchmark, is most suitable for fine-grain TLP using strands. The loads to the C array $scan$ and $match$ can be overlapped. Figure 8 (b) and (c) show the code as partitioned by eBUG for a 2-core system; this results in a speedup of 1.2.

If most instructions have predictable latencies and cache misses are infrequent, ILP is exploited in the coupled mode because it provides the lowest communication latency. An example more suitable for traditional ILP comes from gsmdecode. The code shown in Figure 9 has abundant ILP, and achieves a speedup of 1.78 on a 2-core system.

5 Experimental Evaluation

5.1 Experimental Setup

Our experimental setup for evaluating the Voltron architecture includes a compiler and a simulator built using the Trimaran system [26]. The compiler implements techniques to exploit ILP, fine-grain TLP, and statistical LLP described in Section 4. The multicore simulator models VLIW cores, the dual-mode interconnect network, and coherent caches. The processor model utilizes the HPL-PD instruction set [8] and the latencies of the Itanium processor are assumed. The dual-mode interconnect is modeled to simulate both direct and queue mode communication. We assume a three cycle minimum latency for queue mode (2 cycles plus 1 cycle per hop) and a one cycle minimum latency for direct mode (1 cycle per hop). Large portions of the M5 simulator [1] are utilized to perform detailed cache modeling, incorporating the full effects of maintaining cache coherence. Voltron is assumed to be a bus-based multiprocessor where coherence is maintained by snooping on a shared bus using the MOESI protocol.

We evaluate a two-core and a four-core Voltron system against a single core baseline VLIW machine. In all experiments, each core is a single-issue processor. Each core has a 4 kB 2-way associative L1 D-cache, and a 4 kB 2-way associative L1 I-cache. They share a 128 kB 4-way associative L2 cache.

Performance is evaluated on a subset of benchmarks from the MediaBench and SPEC 2000 suites. Not all benchmarks could be included due to compilation problems in Trimaran.

5.2 Results

Exploit types of parallelism individually. We first evaluate the speedup achieved by exploiting ILP, fine-grain TLP, and LLP separately using the techniques presented in previous sections. ILP is exploited under coupled mode, while LLP and fine-grain TLP are extracted for decoupled mode.

Figure 10 shows the speedup of benchmarks on a 2-core Voltron system. The baseline, 1.0 in the figure, represents the execution time of the benchmark on a single-core processor. Three bars are shown for each benchmark. The first bar represents the speedup achieved by exploiting ILP under coupled mode. The second bar shows the speedup achieved by exploiting fine-grain TLP, using both eBUG and DSWP, under decoupled execution mode. The third bar shows the speedup for benchmarks by exploiting statistical LLP. On average, we achieved speedups of 1.23, 1.16 and 1.18 by individually exploiting ILP, fine-grain TLP and LLP, respectively. As can be seen from the figure, each benchmark best exploits different types of parallelism. 12 out of 25 benchmarks achieved best speedup by exploiting ILP; those benchmarks benefit most from the low latency inter-core communication and synchronized execution. 6 of 25 benchmarks perform best by exploiting fine-grain TLP. They are generally benchmarks with a large number of cache misses. The

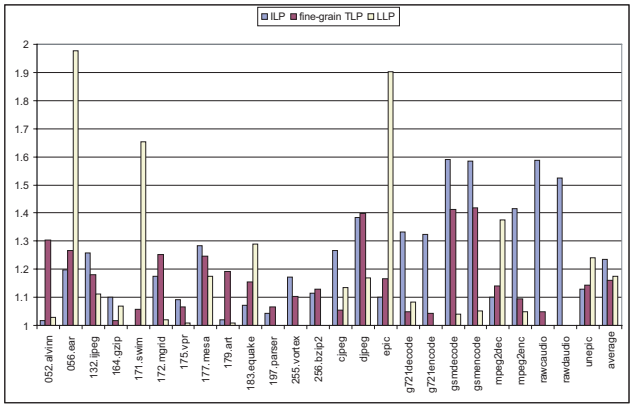

Figure 10: Speedup on 2-core Voltron system exploiting ILP, fine-grain TLP and LLP separately.

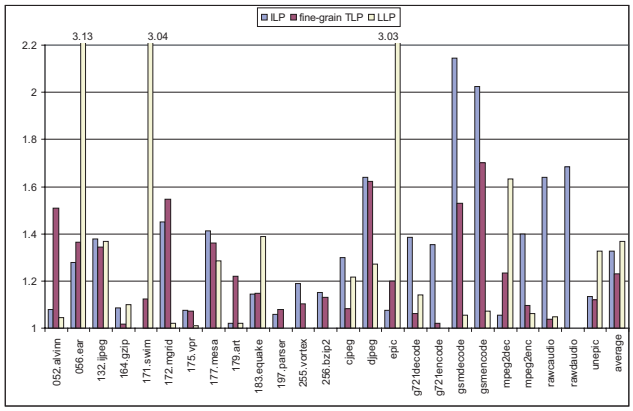

Figure 11: Speedup on 4-core Voltron system exploiting ILP, fine-grain TLP and LLP separately.

decoupled execution allows stalls on one core to overlap with computation or stalls on other cores, thus providing the benefit of coarse-grain out-of-order execution. The 179.art benchmark is representative of this type. The remaining 7 benchmarks have highest speedups when exploiting statistical LLP. Note that many SpecInt and Mediabench programs that traditionally do not show any meaningful loop level parallelism now achieve speedup in Voltron. This is because Voltron supports speculative parallelization of loops that cannot be proven as DOALL. Besides, the low latency communication and synchronization allow parallelization of loops with small trip counts.

Figure 11 shows the same data for a 4-core Voltron system. Similar trends can be observed from this figure. Benchmarks with frequent memory stalls perform better by exploiting fine-grain TLP, benchmarks with statistical DOALL loops benefit from exploiting LLP, and other benchmarks benefit from the low inter-core communication latency of coupled mode by exploiting ILP. In general, the performance gains going from two to four cores is larger for benchmarks that can take advantage of decoupled mode. On average, the speedup for exploiting ILP is 1.33, fine-grain TLP is 1.23, and LLP is 1.37. As can be seen from the data, the speedup achieved from a single execution mode is limited. Many applications cannot be efficiently accelerated by exploiting a single type of parallelism exclusively. Rather, they contain significant code regions that are best suited to each type of parallelism, so the architecture needs to support efficient exploitation of all three types of parallelism.

Memory and communication stalls. Figure 12 shows the relative execution time each benchmark spends on stalls under coupled and decoupled mode when executing on a 4-core Voltron system.[1] Stall cycles are normalized to the total serial execution time. Two bars are shown for each benchmark. The first bar shows the stall cycles when exploiting ILP across all 4 cores (under coupled mode), and the second shows the average stall cycles when exploiting fine-grain TLP (under decoupled mode). Each bar in the graph has several parts representing the various reasons the cores can stall. The bottom-most two parts of each bar show the average stalls due to instruction and data cache misses. For decoupled mode, they are followed by stalls due to receive instructions and synchronization before function calls and returns. The receive stalls are further separated into data receives and predicate receives to study how much stalling is caused by control synchronization.

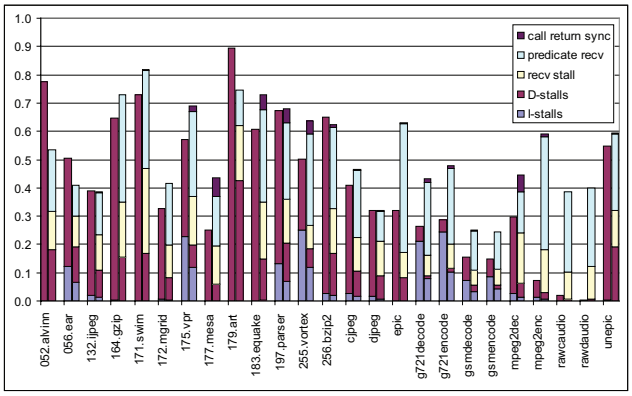

Figure 12: Breakdown of synchronization stalls by type on a 4-core system. Each benchmark has two bars: the left bar is for ILP (coupled mode) and the right for fine-grain TLP (decoupled mode).

As the data indicates, decoupled mode always spends less time on cache miss stalls because different cores are allowed to stall separately. On average, the number of stall cycles under decoupled mode is less than half of that under coupled mode. This explains why it is beneficial to execute memory intensive benchmarks in decoupled mode. However, decoupled mode suffers from extra stalls due to scalar communication and explicit synchronization, which hurts the performance if such communication and synchronization is frequent.

Exploiting hybrid parallelism on Voltron. Figure 13 shows the speedup for benchmarks exploiting all three types of parallelism utilizing both coupled and decoupled modes. The compiler selects the best type of parallelism to exploit for each block in the code. The results show that the hybrid parallelism is much more effective at converting the available resources into performance gain than any individual type of parallelism. For example, the speedup achieved for cjpeg is 1.3 for ILP, 1.08 for fine-grain TLP and 1.21 for LLP on a

[1]Note, for this experiment, the data for decoupled mode only includes fine-grain TLP (i.e., it excludes LLP) as there is virtually no synchronization when executing statistical DOALL loops.

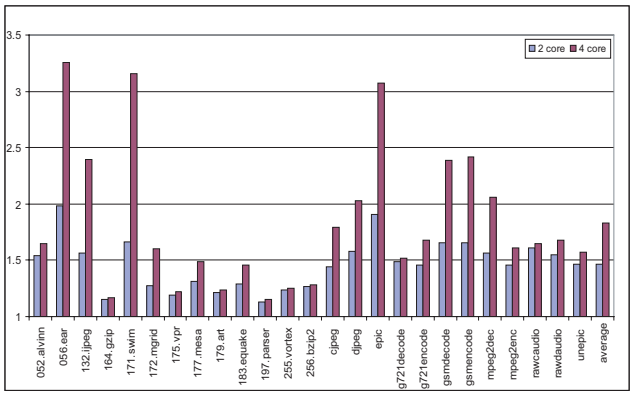

Figure 13: Speedup on 2-core and 4-core Voltron exploiting hybrid parallelism.

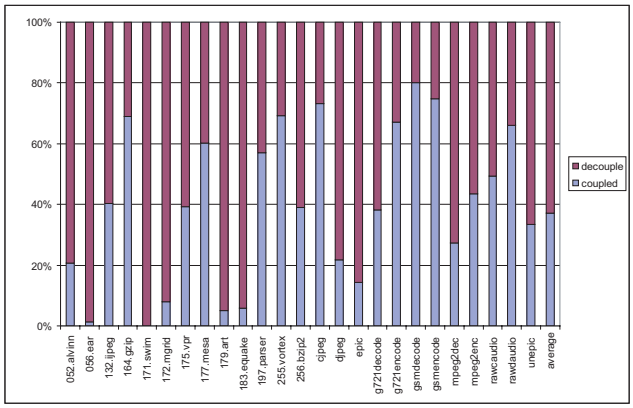

Figure 14: Breakdown of the time spent in each execution mode.

4-core Voltron. As a hybrid of different types of parallelism are efficiently exploited in different parts of the program, a speedup of 1.79 on cjpeg is achieved with dual-mode execution. Overall, the speedup for a 2-core system ranges from 1.13 to 1.98, with an average of 1.46, and the speedup for 4 cores ranges from 1.15 to 3.25, with an average of 1.83.

During hybrid execution, the architecture spends significant amounts of time in both coupled and decoupled modes. Figure 14 shows the percentage of time spent in each mode. Applications such as epic, which has abundant fine-grain TLP, spends most of its time in decoupled mode. Other applications, such as cjpeg, truly take advantage of the mixed modes. As shown earlier in Figure 3, cjpeg has quite a bit of LLP, but another significant portion of the program is best suited for ILP. Hybrid execution allows this and other benchmarks to achieve better speedup than they would in either mode.

6 Related Work

Several architectures have been proposed to exploit fine-grain parallelism on distributed, multicore systems. RAW [24] is the most similar to this work and is a general-purpose architecture that supports instruction, data and thread-level parallelism. In RAW, a set of single-issue cores are organized in a two dimensional grid of identical tiles. Scalar operand networks are used to route intermediate register values between tiles [25]. Both the execution on tiles and the routing on the network are software controlled by the compiler. A dynamic routing network also exists to route memory values. The central difference between Voltron and RAW is the dual execution mode capability and its associated effects on the architecture. The two execution modes allow Voltron to exploit fine-grain TLP like RAW as well as VLIW-style ILP.

The M-Machine [5] is a multiprocessor system that focuses on exploiting fine-grain parallelism. One unique feature of the M-Machine is the direct inter-processor communication channels. Processors communicate intermediate results by directly writing values into the register files of other processors. The M-Machine can exploit both ILP and TLP using these mechanisms. Voltron assumes a more traditional organization of the individual cores and layers on top of that the dual-mode operand network. Further, the lack of communication queues in the M-Machine limits the amount of decoupling between threads. The M-Machine also requires explicit synchronization between processors to exploit ILP, while coupled mode in Voltron allows implicit synchronization for lower-latency communication. The J-Machine [16] is a multicomputer that supports fast message passing and synchronization between nodes. It showed that fine-grain parallel computing is feasible on multiprocessors and serves as strong motivation for this work.

Distributed processing has also been investigated in the context of superscalar processors, including Multiscalar [23], Instruction Level Distributed Processing (ILDP) [10], and TRIPS [22]. In the Multiscalar processor, a single program is divided into a collection of tasks by a combination of software and hardware. The tasks are distributed to multiple parallel processing units at runtime and execute speculatively. Recovery mechanisms are provided if the speculation is wrong. The ILDP architecture contains a unified instruction fetch and decode; the front-end distributes strands of dependent operations to multiple distributed PEs. An interconnection network connects the PEs and routes register values. TRIPS supports explicit data graph execution, which is similar to dataflow execution, to exploit ILP from single thread programs. The execution units on TRIPS are hyperblocks. At runtime, the cores fetch blocks from memory, execute them and commit the results to architectural state. Decoupled execution in Voltron is different from these architectures in that program execution is fully orchestrated by the compiler. The compiler partitions the code, assigns code to cores, schedules the execution within each core and inserts communication between cores. The compiler orchestrated execution allows the hardware to be simple and efficient.

Multicluster VLIW architectures utilize distributed register files and processing elements to exploit ILP. The Multiflow TRACE/500 VLIW architecture [3] contains two replicated sequencers, one for each 14-wide cluster. The two clusters can execute independently or rejoin to execute a single program. MultiVLIW [21] is a distributed multicluster VLIW with architecture/compiler support for scalable distributed data memories. The XIMD is a VLIW architecture that can partition its resources to support the concurrent execution of a dynamically varying number of instruction streams [27]. Voltron extends these works with decoupled execution to exploit LLP and fine-grain TLP.

Dual-core execution [29] proposes mechanisms to accelerate single thread application through runahead execution [15] on two cores. In dual-core execution, a front processor runs ahead to warms up caches and fixes branch mispredications for a back processor. This approach provides

benefit for memory intensive applications but is not very scalable to multiple cores.

7 Conclusion

This paper proposes a multicore architecture, referred to as Voltron, that is designed to exploit the hybrid forms of parallelism that can be extracted from general-purpose applications. The unique aspect of Voltron is the ability to operate in one of two modes: coupled and decoupled. In coupled mode, the cores execute in lock-step forming a wide-issue VLIW processor. Each core executes its own instruction stream and through compiler orchestration, the streams collectively execute a single thread. In decoupled mode, the cores operate independently on fine-grain threads created by the compiler. Coupled mode offers the advantage of fast inter-core communication and the ability to efficiently exploit ILP across the cores. Decoupled mode offers the advantage of fast synchronization and the ability to overlap execution across loop iterations and in code regions with frequent cache misses.

The performance evaluation shows that the dual mode capabilities of Voltron are highly effective. Voltron achieves an average performance gain of 1.46x on a 2 core system and 1.83x on 4 core system over a single core baseline. Dual-mode execution is significantly more effective than either mode alone. The ability to switch between coupled and decoupled modes across different code regions allows the architecture to adapt itself to the available parallelism. This behavior reflects the disparate application characteristics, including cache miss rates and forms of available parallelism, that are common in general-purpose applications.

8. Acknowledgments

We thank Mike Schlansker for his excellent comments and suggestions for this work. Much gratitude goes to the anonymous referees who provided helpful feedback on this work. This research was supported by the National Science Foundation grant CCR-0325898, the MARCO Gigascale Systems Research Center, and equipment donated by Hewlett-Packard and Intel Corporation.

References

[1] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-oriented full-system simulation using M5. In *6th Workshop on Computer Architecture Evaluation using Commercial Workloads*, pages 36–43, Feb. 2003.

[2] M. Chu, K. Fan, and S. Mahlke. Region-based hierarchical operation partitioning for multicluster processors. In *Proc. of the SIGPLAN '03 Conference on Programming Language Design and Implementation*, pages 300–311, June 2003.

[3] R. Colwell et al. Architecture and implementation of a VLIW supercomputer. In *Proc. of the 1990 International Conference on Supercomputing*, pages 910–919, June 1990.

[4] J. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, Cambridge, MA, 1985.

[5] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee. The m-machine multicomputer. In *Proc. of the 28th Annual International Symposium on Microarchitecture*, pages 146–156. IEEE Computer Society Press, 1995.

[6] J. Fisher. Very Long Instruction Word Architectures and the ELI-52. In *Proc. of the 10th Annual International Symposium on Computer Architecture*, pages 140–150, 1983.

[7] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[8] V. Kathail, M. Schlansker, and B. Rau. HPL-PD architecture specification: Version 1.1. Technical Report HPL-93-80(R.1), Hewlett-Packard Laboratories, Feb. 2000.

[9] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[10] H. Kim and J. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *Proc. of the 29th Annual International Symposium on Computer Architecture*, pages 71–81, June 2002.

[11] D. J. Kuck. *The Structure of Computers and Computations*. John Wiley and Sons, New York, NY, 1978.

[12] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages 81–92, Dec. 2003.

[13] R. Leupers. *Code Optimization Techniques for Embedded Processors - Methods, Algorithms, and Tools*. Kluwer Academic Publishers, Boston, MA, 2000.

[14] S. A. Lieberman, H. Zhong, and S. A. Mahlke. Extracting statistical loop-level parallelism using hardware-assisted recovery. Technical report, Department of Electrical Engineering and Computer Science, University of Michigan, Feb. 2007.

[15] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. In *Proc. of the 9th International Symposium on High-Performance Computer Architecture*, pages 129–140, Feb. 2003.

[16] M. D. Noakes, D. A. Wallach, and W. J. Dally. The j-machine multicomputer: an architectural evaluation. In *Proc. of the 20th Annual International Symposium on Computer Architecture*, pages 224–235. ACM Press, 1993.

[17] E. Nystrom and A. E. Eichenberger. Effective cluster assignment for modulo scheduling. In *Proc. of the 31st Annual International Symposium on Microarchitecture*, pages 103–114, Dec. 1998.

[18] E. Nystrom, H.-S. Kim, and W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proc. of the 11th Static Analysis Symposium*, pages 165–180, Aug. 2004.

[19] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 105–118, Nov. 2005.

[20] E. Özer, S. Banerjia, and T. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proc. of the 31st Annual International Symposium on Microarchitecture*, pages 308–315, Dec. 1998.

[21] J. Sanchez and A. Gonzalez. Modulo scheduling for a fully-distributed clustered VLIW architecture. In *Proc. of the 33rd Annual International Symposium on Microarchitecture*, pages 124–133, Dec. 2000.

[22] K. Sankaralingam, R. Nagarajan, H. Liu, J. Huh, C. Kim, D. Burger, S. Keckler, and C. Moore. Exploiting ILP, TLP, and DLP using polymorphism in the TRIPS architecture. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, June 2003.

[23] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[24] M. Taylor et al. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, pages 2–13, June 2004.

[25] M. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal. Scalar operand networks: On-chip interconnect for ILP in partitioned architectures. In *Proc. of the 9th International Symposium on High-Performance Computer Architecture*, pages 341–353, Feb. 2003.

[26] Trimaran. An infrastructure for research in ILP, 2000. http://www.trimaran.org/.

[27] A. Wolfe and J. Shen. A variable instruction stream extension to the VLIW architecture. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–14, Oct. 1991.

[28] H. Zhong, K. Fan, S. Mahlke, and M. Schlansker. A distributed control path architecture for VLIW processors. In *Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 197–206, Sept. 2005.

[29] H. Zhou. Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window. In *Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 231–242, Sept. 2005.