A Distributed Control Path Architecture for VLIW Processors

Hongtao Zhong[1], Kevin Fan[1], Scott Mahlke[1], and Michael Schlansker[2]

[1]Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
{hongtaoz, fank, mahlke}@umich.edu

[2]Hewlett Packard Laboratories
Palo Alto, CA
{schlansk}@hpl.hp.com

ABSTRACT

VLIW architectures are popular in embedded systems because they offer high-performance processing at low cost and energy. The major problem with traditional VLIW designs is that they do not scale efficiently due to bottlenecks that result from centralized resources and global communication. Multicluster designs have been proposed to solve the scaling problem of VLIW datapaths, while much less work has been done on the control path. In this paper, we propose a distributed control path architecture for VLIW processors (DVLIW) to overcome the scalability problem of VLIW control paths. The architecture simplifies the dispersal of complex VLIW instructions and supports efficient distribution of instructions through a limited bandwidth interconnect, while supporting compressed instruction encodings. DVLIW employs a multicluster design where each cluster contains a local instruction memory that provides all intra-cluster control. All clusters have their own program counter and instruction sequencing capabilities, thus instruction execution is completely decentralized. The architecture executes multiple instruction streams at the same time, but these streams collectively function as a single logical instruction stream. Simulation results show that DVLIW processors reduce the number of cross-chip control signals by approximately two orders of magnitude while incurring a small performance overhead to explicitly manage the instruction streams.

1. INTRODUCTION

Many embedded systems perform computationally demanding processing of images, sound, video, or packet streams. VLIW architectures are popular for such systems because they offer the potential for high-performance processing at a relatively low cost and energy usage. In comparison to ASIC solutions, VLIWs are programmable, and therefore can support multiple applications or software changes. Several examples of VLIW designs include the TI C6x series, Lx/ST200, and Philips TM1300.

A major challenge with traditional VLIW processors is that they do not scale effectively or efficiently due to two major problems: centralized resources and wire delay. Centralized resources, including the register file and instruction decode logic, become the cost, energy, and delay bottlenecks in a VLIW design as they are scaled to support more function units (FUs). As feature sizes decrease, wire delays are growing relative to gate delays. This has a serious impact on processor designs as distributing control and data each cycle takes more time and energy [22, 4, 32]. Wire delays are further exacerbated when the processor width is scaled as the distance between FUs, register files and caches increases, thereby forcing the signals to travel further.

To support efficient scaling of VLIW datapaths, multicluster designs have been proposed. In a multicluster design, the centralized register file is broken down into several smaller register files. Each of the smaller register files supplies operands to a subset of the FUs, known as a *cluster* [12, 7]. A design can be efficiently scaled by adding more clusters as the bottlenecks produced by the centralized register file are removed. The clustering approach can be similarly applied to the data memory subsystem. Data caches can be partitioned and distributed to each cluster in the form of hardware-managed caches [25, 13] or software-managed buffers [14].

VLIW processors face a similar scaling problem with the control path where conventional designs utilize a centralized instruction memory or cache to store instructions. A centralized instruction fetch, decode, and distribution system issues control signals on every clock cycle to all the FUs and storage elements in the datapath to direct their operation. This centralized control system does not scale well due to complexity, latency, and energy consumption. As processor issue width is scaled, the number of instruction bits grows accordingly, increasing hardware cost for instruction fetch, decode, and distribution. Variable-length instruction encodings can be used to reduce code size, but often make the problem worse by increasing the complexity of the instruction alignment and distribution networks [3]. The distance separating FUs and storage elements from the instruction memory also grows as the design is scaled, thereby exposing the wire delay problem in the same manner as in the datapath.

In this paper, we introduce a distributed control path architecture for VLIW processors called distributed VLIW, or DVLIW, to support scalable control path design. The architecture simplifies the dispersal of complex VLIW instructions and supports efficient distribution of instructions through local interconnects. These goals are achieved without sacrificing the code size benefits of compressed instruction encodings. The central idea is to distribute the instruction fetch, decode, and distribution logic in the same manner that the register file is distributed in a multicluster datapath. DVLIW has a multicluster datapath consisting of an array of clusters. Each cluster contains an instruction memory or cache combined with fetch, decode, and distribution units that provide control within a cluster. All clusters have their own program counter (PC) and next PC generation hardware to facilitate distributed instruction sequencing.

With DVLIW, multiple instruction streams are executed at the same time. These streams collectively function as a single logical stream on a conventional VLIW processor. Although clusters fetch independently, they execute operations in the same logical position each cycle and branch to the same logical location. Procedures, basic blocks, and instruction words are vertically sliced and stored in different locations in the instruction memory hierarchy. The logical organization is maintained by the compiler to ensure proper execution. The DVLIW architecture can be viewed as a special chip multiprocessor system that, through compiler orchestration, collectively executes a single program exploiting ILP. A DVLIW architecture can also dynamically repartition itself to support concurrent execution of multiple instruction streams. Thus, applications with both large degrees of parallelism as well as those with limited but thread-level parallelism can be efficiently executed on the DVLIW.

The DVLIW architecture derives its roots from the Multiflow TRACE and XIMD architectures [7, 33]. The Multiflow TRACE/500 VLIW architecture contains two replicated sequencers, one for each 14-wide cluster. The two clusters can execute independently (separate threads) or rejoin to execute a single program. The XIMD architecture generalized and formalized this concept. An XIMD processor has multiple control units. Several control units can operate identically, emulating a VLIW processor, or can partition the processor resources to support the concurrent execution of multiple instruction streams. Partitioning can vary dynamically to allow for efficient execution of parallel loops [23]. Both the Multiflow TRACE/500 and XIMD can execute one or more programs concurrently with distributed control. These machines use a combined software/hardware strategy that emulates a VLIW using a common but replicated program counter for all clusters. When no instruction compression is utilized, programs within each cluster take an identical amount of space. This allows simple branch strategies that synchronize program counters across clusters.

The net effect of replicating a centralized PC is that it inherently limits the use of any form of compressed instructions, particularly variable length encodings. Each instruction sequencer must behave identically, thus each cluster must have a constant instruction rate. NOPs must be inserted to ensure the instruction words are fully expanded and of constant size. Further, variable length encodings for different operation types (e.g., move versus add, or the use of different literal sizes) cannot be used. One of the central contributions of DVLIW is the combination of architectural and compiler support to distribute the PC while supporting flexible instruction compression technology in order to provide efficient usage of the instruction memory.

2. BACKGROUND AND RELATED WORK

A large body of prior work exists in the area of VLIW processor design and architectures for supporting distributed execution. We focus the discussion on the two areas most related to DVLIW: architecture models for distributed ILP processing and VLIW instruction memory systems.

Distributed Architecture Models. DVLIW is built directly upon prior multicluster VLIW processor designs, including Multiflow TRACE [7], XIMD [33], and MultiVLIW [25]. The clustered register file was first introduced in the Multiflow processor to facilitate wide-issue design. MultiVLIW has expanded this work by focusing on alternative architecture/compiler strategies for designing scalable distributed data memory subsystems. Interleaved data caches [13] or compiler-managed L0 buffers [14] independently supply memory data to each cluster. DVLIW extends these works in an orthogonal direction by focusing on fully distributing the control path.

There are a number of parallels between DVLIW and the Raw architecture [31]. Raw is a general-purpose architecture that effectively supports instruction, data, and thread-level parallelism. In Raw, all processor resources are fully distributed in the form of a two dimensional grid of identical tiles. Further, all processor resources are software-controlled and managed by the compiler. Scalar operand networks are used to effectively route intermediate register values from producer to consumer tiles [30]. Our work shares the central concept of software-controlled distributed ILP processing. However, the primary difference is that Raw is organized more like a CMP. All tiles run independently and communicate through message queues and routers. This organization is inherently more adaptable, but with a higher hardware cost. Conversely, DVLIW is more geared to embedded systems. It supports fewer, but wider clusters with all clusters executing in lock step, thereby creating a more energy-efficient design.

Distributed processing has also been investigated in the context of superscalar processors including Multiscalar [28], Instruction Level Distributed Processing (ILDP) [16] and TRIPS [26].

VLIW Instruction Memory Systems. The second area of prior work is the design of VLIW instruction memory systems. The design space is taxonomized on two independent axes: binary encoding style and degree of centralization.

The first characteristic that differentiates VLIW instruction memory systems is the binary encoding style. VLIW instructions can be encoded using either an uncompressed or compressed representation. Although many hybrid compression strategies involving NOPs and variable width operations are possible, only the two extremes of NOP compression are considered here to simplify the discussion. An uncompressed encoding is one that explicitly stores NOPs for a particular FU when it is idle in a cycle. Conversely, compressed encodings avoid explicitly storing NOPs to reduce code size. NOPs in the code can be compressed both horizontally and vertically. Horizontal compression reduces NOPs within a VLIW instruction, while vertical compression removes entire cycles of NOPs between instructions.

Several compressed encodings for VLIW processors have been proposed. The TINKER encoding accomplishes horizontal compression by using Head and Tail bits within an operation to delineate the beginning and end of an instruction [8]. Every operation also contains a Pause field to indicate the number of empty instructions after this instruction, thereby accomplishing vertical compression. The Cydra 5 [24], PICO VLIW [3], and the Intel Itanium employ multiple instruction templates to accomplish horizontal compression. Each template provides operation slots for a subset of the FUs; the compiler selects the closest template to remove the majority of NOPs. In addition, many techniques not only avoid explicit NOPs but also reduce the size of useful operations. For example, data compression methods are used to compress code in embedded systems [17, 21, 34]. Larin and Conte presented techniques for reducing VLIW code size by Huffman compression or tailored encoding of the instruction set [18].

The second characteristic that differentiates VLIW instruction memory systems is whether a centralized or distributed instruction memory is utilized. A centralized instruction memory stores operations for all FUs. Conversely, with the distributed approach, multiple instruction memories are created with each storing operations for a subset (cluster) of the FUs.

The cross product of both characteristics defines four high-level VLIW instruction memory configurations: centralized/uncompressed, centralized/compressed, distributed/uncompressed, and distributed/compressed as shown in Figure 1. Centralized/uncompressed, Figure 1(a), provides the simplest organization in that the instruction word length and the position of operations for each FU are fixed, thus the mapping from I-cache to instruction register (IR) is trivial. However, for a wide issue processor, the I-cache utilization can be extremely low. To remove the NOPs, centralized/compressed, Figure 1(b), increases the complexity of the fetch unit as it needs to partially decode the instruction to determine the boundary of the instruction word and correctly expand the compressed instruction

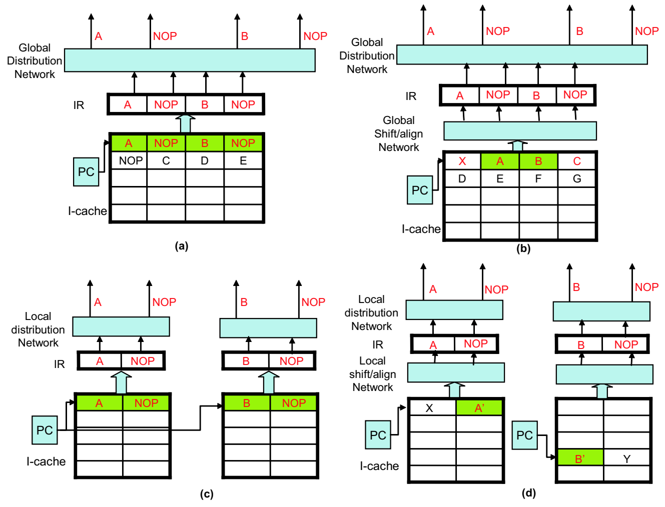
A
NOP
B
NOP
Global Distribution Network
IR
PC
I-cache
(a)
Global Shift/align Network
X
C
D
E
F
G
(b)
Local distribution Network
(c)
Local shift/align Network
A'
B'
Y
(d)

Figure 1: Instruction memory organizations: (a) Centralized instruction cache, uncompressed encoding; (b) Centralized instruction cache, compressed encoding; (c) Distributed I-cache, centralized PC, uncompressed encoding; (d) Distributed I-cache, distributed PC, compressed encoding.

into the IR.

The scalability problem of centralized I-caches can be overcome by utilizing distributed caches. With a distributed approach in a multicluster processor, each cluster has its own I-cache to supply instructions. Assuming each cluster is homogeneous, all operations are the same size, and no compression, all clusters execute the same cache entry each cycle. Thus, the distributed/uncompressed (Figure 1(c)) can utilize a common PC to index into each I-cache. The use of a common PC relies on each cluster having a constant instruction rate during the execution. Most previous research on distributed instruction memories falls into this category as they rely on a single PC or multiple identical PCs to index the memories when a single program is executed. Since XIMD has no I-cache, a more sophisticated compiler may be able to remove this restriction [33]. Another approach, the Silo cache [8], partitions the instruction cache into silos, where each silo holds operations for a particular set of FUs. All silos are indexed by a single PC. Although instructions must be uncompressed, multiple instructions can coexist at the same address across silos.

Distributed/compressed, Figure 1(d), is the proposed DVLIW instruction memory organization. To support compressed encodings with a distributed I-cache, the PC must also be distributed as instruction sizes for each cluster must be allowed to vary. All the clusters fetch independently from different PCs and compute their own next PCs. The DVLIW architecture and its operation are discussed in detail in the following sections.

3. DVLIW ARCHITECTURE

3.1 Overview

The DVLIW architecture completely decentralizes the instruction memory to facilitate distributed instruction fetch, alignment, decode, distribution, and sequencing. Together with other clustering techniques, the distributed instruction memory allows operations to be fetched and executed locally within a cluster. The primary advantage of this design is that it is both more scalable and hardware efficient, since the bottleneck of the traditional centralized instruction memory is removed. Furthermore, full support for compressed encodings in all levels of memory hierarchy is provided by the architecture to maintain small code size. Thus, the architectural model is particularly appropriate for embedded systems.

Figure 2 presents a block diagram of a four cluster DVLIW processor. Clusters are organized in a one or two dimensional array, surrounded by a banked L2 cache. Such an organization is similar to tiled architectures, such as Raw [31]. Inter-cluster data communication is handled via connections between neighboring clusters. The latency of inter-cluster communication is exposed to the compiler, which schedules explicit ICMOVE (inter-cluster move) operations to transfer data between clusters. A one-bit bus connecting all clusters is used for propagating stall signals. This globally propagated stall signal maintains synchronization among DVLIW clusters. A second bus is used for broadcasting branch conditions and will be described later in this section.

The datapath of each cluster is similar to that of a conventional multicluster VLIW, with each cluster having its own FUs, general purpose register file (GPR), predicate register file (PR), branch target register file (BTR), and floating point register file (FPR), and a inter-cluster move unit to transfer data between neighbor clusters. Each cluster also has its own data cache; a bus-based snooping protocol is assumed to maintain coherence among data caches. The DVLIW architecture does not limit the way in which data caches are organized—any hardware or software coherence protocol will suffice. In addition, the control path is fully partitioned as each cluster has its own program counter (PC), instruction cache, shift/align network, and instruction register (IR). In every cycle, each cluster fetches operations from its I-cache according to its own PC. All clusters execute synchronously and the execution order of operations is the same as that of a traditional VLIW architecture. All operations in a logical instruction word are fetched and executed at the same cycle in different clusters. If any cluster incurs a cache miss, all clusters must stall.

Within each cluster's instruction cache, there are no restrictions on the code compression techniques that may be used. Compression schemes, such as TINKER, instruction templates, or Huffman code based techniques as described in Section 2, may be used to preserve small code size while distributing the instructions. The compressed instructions are decompressed as they move from the I-cache to the IR in each cluster via the shift/align network.

The code organization in all levels of the memory hierarchy is changed for DVLIW. In conventional architectures with a centralized PC, all operations within the same instruction word are placed sequentially in memory, as shown in Figure 3(a). Thus, operations for different clusters, e.g. A0 and A1, are placed next to each other. With such a code organization, distributing the I-cache is difficult because the actual distribution of instruction bits must occur during program execution by the hardware. This generally precludes any instruction compression schemes as the run-time distribution algorithm must be very simple. In the DVLIW architecture, the operations for each cluster are grouped together, and code for different clusters is placed separately in the memory (or in separate memories), as shown in Figure 3(b). This organization of code allows a cluster to compute its own next PC without knowing the size of operations in other clusters, thus allowing all clusters to fetch and execute independently.

3.2 Branch Mechanism

A major challenge that arises for the DVLIW architecture is branch execution. A valid single thread of execution must be maintained using multiple instruction streams. Therefore, each instruction stream must execute instructions from the same logical instruc-

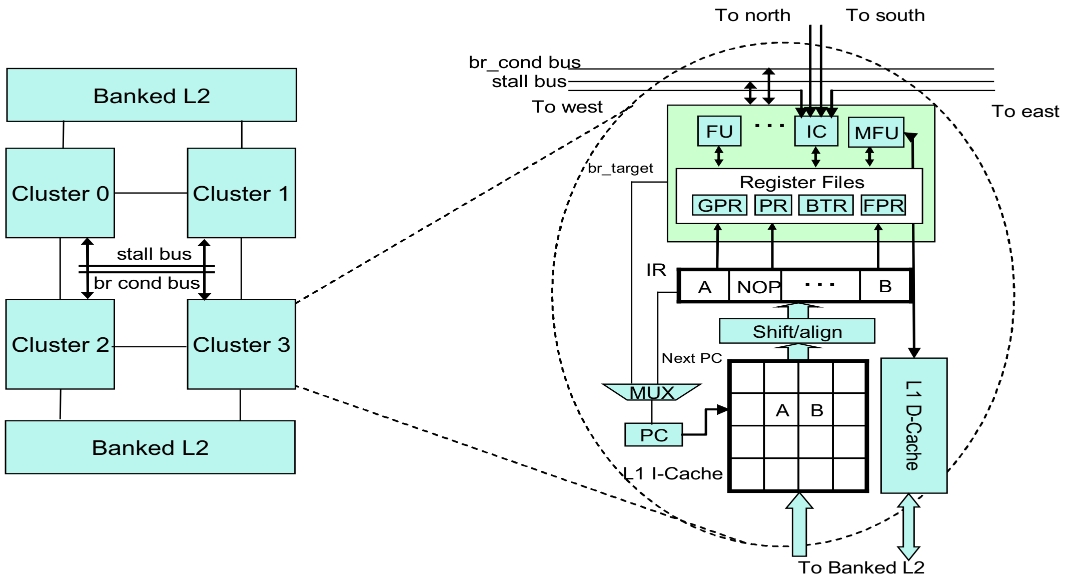

Figure 2: DVLIW architecture overview. Four cluster example is shown.

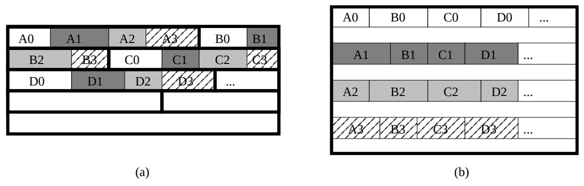

Figure 3: (a) Code layout on a conventional 4-cluster VLIW architecture. Rectangle A0 represents operations in MultiOp A to be executed by cluster 0. (b) Code layout on the DVLIW architecture. Bold lines represent MultiOp boundaries in (a).

tion word each cycle and branch to the same logical target at the same time. As shown in Figure 3(b), operations in a logical instruction word are stored in separate locations. Thus, every cluster has a different branch target for each branch operation. Special architectural and compiler support is proposed to solve this problem.

The proposed branch mechanism is based on the unbundled branch architecture in HPL-PD [15]. The unbundled branch architecture separately specifies each portion of the branch: target address, condition, and control transfer point. In this manner, each portion of a branch can be specified as soon as it becomes available to reduce the latency associated with branches. For example, the branch target is static and can be specified well in advance of the branch to permit instruction prefetching. In the HPL-PD architecture, each portion of the branch is performed by a separate operation. The operations involved in branches are as follows:

Branch target address specification. Prepare-to-branch (PBR) operations are used to specify the target address and a static prediction for a branch ahead of a branch point, typically initiating instruction prefetch when the prediction bit is set to taken.

Branch condition computation. Compare-to-predicate (CMPP) operations are used to compute branch conditions, which are stored in the PR file.

Control transfer. The actual branch operations test the branch condition and perform the transfer of control to the target address. Control transfer operations include branch-if-condition-true (BRCT) and branch unconditionally (BRU).

There is no dependence between the computation of the branch target address and the computation of the branch condition, so the order of the first two steps is not restricted. An unconditional branch does not need the computation of the branch condition, and thus consists of only the first and last steps.

In a DVLIW architecture with unbundled branches, branch targets must be computed separately for each cluster. The idea of an *external branch* is introduced to represent the implementation of a branch on another cluster. For each original branch scheduled on a cluster, an external branch on all other clusters must be created by compiler to ensure that proper control flow is maintained in each instruction stream. As a result, the HPL-PD unbundled branch process becomes slightly more complex. Four steps are required as follows:

Branch target address specification. To specify a separate branch target for each cluster, a separate PBR operation must be issued for each cluster. A conventional PBR operation is used for the cluster with the original branch, and an external PBR (EPBR) is created for each of the other clusters. Both the PBR and EPBRs behave identically to the original PBR, storing the branch target in a BTR register and initiating a prefetch if the prediction bit is set. The external distinction is only a logical one. The targets of the PBR and EPBRs correspond to the same logical block. The actual physical address for each is filled in by subsequent assembling and linking after all of the code is fully bound.

Branch condition computation. There is no change in the branch condition computation. A single CMPP operation is issued on a cluster and the result is stored in the predicate register file. Unlike the branch target address which is different for each cluster, the branch condition is the same for all clusters.

Branch condition distribution. In the DVLIW architecture, each cluster containing an external branch must be informed of the branch condition. A broadcast operation, BCAST, is executed by the cluster where the branch condition was computed. This operation is an inter-cluster move with one source and multiple destinations; it copies the branch condition bit from the predicate register file where it was computed to the predicate register files of the other clusters. In order to save encoding size, the multiple destinations have the same register number. (Alternately, the compiler can insert a branch condition receive operation in all clusters with an external branch, thereby broadcasting the condition bit to arbitrary register locations in each cluster).

time	Cluster 0		Cluster 1	
	slot 0	slot 1	slot 0	slot 1
0	ICMOVE r34, r17	SHRA r27, ret, 31	MOVE r5, _s	
1	MOVE r18, 1	AND r17, r27,1		LD r4, r34
2	ADD r20, r17, ret	PBR BTR6, BB42	EXTS r9, r4	
3	ICMOVE r6, r21	SHRA r24, r20,1		
4	CMPP PR7, r24<=0	MOVE r10, _abuf	SHL r7, r6, 2	
5	ADD r10, r10, r11			LD r8, r7
6	BRCT BTR6, PR7			

(a)

Cluster 0			Cluster 1	
slot 0	slot 1		slot 0	slot 1
ICMOVE r34, r17	SHRA r27, ret, 31		MOVE r5, _s	**EPBR BTR4,BB42**
MOVE r18, 1	AND r17, r27,1			LD r4, r34
ADD r20, r17, ret	PBR BTR6, BB42		EXTS r9, r4	
ICMOVE r6, r21	SHRA r24, r20,1	…		
CMPP PR7,r24<=0	MOVE r10, _abuf		SHL r7, r6, 2	
ADD r10, r10, r11	**BCAST PR7**			LD r8, r7
BRCT BTR6, PR7			**EBR BTR4, PR7**	

(b)

Figure 4: Example code segment for a two cluster processor: (a) Traditional VLIW, (b) DVLIW. The leftmost operand is the destination for each assembly operation.

Control transfer. If the branch condition is taken, each cluster transfers control to its individual branch target. All of these branch targets correspond to the same logical block. In the cluster with the original branch, a conventional HPL-PD branch operation is used. New branch operations, called external branches (EBR), are used on the other clusters. EBRs behave exactly as BRCT operations (again the naming is for logical purposes only) using the broadcast condition. A conservative approach is that the control transfers must all be scheduled at the same cycle to guarantee correct execution order and enforcement of dependences, as together they implement one branch in a traditional VLIW.

Special support is provided in DVLIW for efficient distributed execution of software-pipelined loops. In HPL-PD, two special registers, the loop count (LC) and the epilogue stage count (ESC), are used to control loop execution. The LC register automatically decreases by one in every loop iteration until it reaches zero, then the ESC is decremented to drain the pipeline. The loop execution condition is determined solely by testing the LC and ESC. With DVLIW, the LC and ESC are replicated in every cluster and initialized in the preloop. Once execution starts, each cluster updates its own LC and ESC. As a result, the cluster control is completely decoupled during the execution of software pipelined loops. Note that inter-cluster data communication is still necessary to transfer register values between clusters in most loops.

3.3 Example

The example in Figure 4 illustrates the organization and operation of DVLIW code. This is a basic block adapted from the rawcaudio benchmark [19]. The code is compiled for a two-cluster processor with each cluster supporting two operations of any type per cycle. The figure shows the VLIW schedule in tabular form with each row comprising a single instruction word. Figure 4(a) shows the code for a traditional multicluster VLIW. The branch at the end of the basic block is realized by three operations: the PBR at cycle 2, the CMPP at cycle 4, and the BRCT at cycle 6. Control is transferred to block 42 if the condition evaluates to True. Note that no branch operations are required on cluster 1.

Figure 4(b) shows the corresponding DVLIW code. There are two major changes in the code. First, the code for the basic block is split into two disjoint pieces, one for each cluster, and they are stored in different memory locations. Second, three new operations (shown in bold in the figure) are inserted. An EPBR is inserted on cluster 1 at cycle 0 to specify the branch target for cluster 1 corresponding to logical block 42. A BCAST is inserted on cluster 0 at cycle 5 to transmit the branch condition contained in predicate PR7 on cluster 0 to PR7 on cluster 1. Finally, an EBR is inserted in cluster 1 and scheduled at the same cycle as the BRCT in cluster 0. The EBR reads the branch condition from the register written by the BCAST, and branches to the specified target if the value is True.

Note that the two clusters enter the basic block at the same time, but with differing local PCs. Before execution of the BRCT and EBR, each cluster executes instructions independently. The only inter-cluster control communication occurs in cycle 5 when the BCAST operation is executed, sending the branch condition from cluster 0 to cluster 1. There are two inter-cluster data communications (ICMOVEs) at cycles 0 and 3.

3.4 Analysis of the Architecture

As global communication in processors becomes critical from cost, latency, and energy efficiency perspectives, the DVLIW architecture has a number of advantages. First, DVLIW can effectively remove the bottleneck caused by global wires to fetch and distribute operations in wide issue machines. In this architecture, the resources needed for instruction fetch, including instruction cache and next PC generation hardware, are localized within the clusters. Together with other datapath and data cache clustering techniques, DVLIW provides an efficient architecture for creating scalable designs. Furthermore, the elimination of global communication reduces control path energy and latency. In addition, DVLIW supports both horizontal and vertical instruction compression within each cluster using any existing technique. The compression of code is not possible in traditional distributed I-cache schemes where the PC is still centralized.

On the negative side, there are some costs associated with DVLIW. The architecture needs extra operations to be inserted for every branch on every cluster, which leads to increased code size. Furthermore, the dependence height of a block can be increased if a branch is on the critical path. Third, a larger number of I-cache misses may occur since each cache is independently managed. Last, the global stall signal bus and the branch condition bus may limit the scalability of the processor. To address these challenges, the next section proposes ways to reduce the code size expansion and the impact of I-cache misses. In addition, issues regarding procedure calls and compiler partitioning are discussed.

4. IMPLEMENTATION ISSUES

4.1 Cluster Sleep Mode

In the DVLIW architecture, multiple clusters collectively execute a single program exploiting instruction/loop level parallelism. Some applications may not have enough parallelism to keep all of the clusters busy, in which case some clusters are idle for portions of the application's execution. Also, a heterogeneous clustered machine may have a cluster dedicated to floating-point operations; when integer code is executed, the floating-point cluster is idle.

In the basic DVLIW model, every cluster must keep its control flow synchronized with other clusters even if the cluster is not doing any useful work. If a cluster branches to a block, other clusters also need to branch to the same logical block at the same time. To keep the idle clusters synchronized with other clusters, every empty block must contain the EPBRs and EBRs, and at least one NOP at the beginning for synchronization. These operations in idle clusters increase static code size and waste energy when they are executed.

A software-controlled mechanism is proposed to handle idle clusters. The idea is to allow the compiler to insert operations to explicitly place a cluster in sleep mode and wake a cluster to resume execution. In sleep mode, the cluster does nothing, but values in the register files are kept unchanged.

To support sleep mode, several new operations are introduced into the instruction set architecture (ISA). In DVLIW, switching between sleep mode and wake mode is always done at a block boundary. Thus, the sleep operation is always executed in the same cycle as a branch. There are three operations that can begin sleep mode: SLEEP_TK, SLEEP_NT, and SLEEP_AL. All of these operations have no operands, so their encoding can be very compact. SLEEP_TK (NT) sets the cluster it is executing on to sleep mode if the branch executing in the same cycle is taken (not taken). SLEEP_AL always sets the cluster to sleep mode whether the branch is taken or not.

Sleep mode could be supported by adding a single SLEEP operation rather than three, and giving the SLEEP operation a predicate operand. We choose to add three different sleep operations instead of one to reduce the encoding size. SLEEP_TK and SLEEP_NT get their predicate implicitly from the branch in the same cycle, so all three of these operations can be encoded as two bits in the encoding of a branch operation. Since most branch operations do not have a destination operand, two bits are available for sleep modifiers, and overall code size is not affected.

Waking up a sleeping cluster is more complicated as it must be done by a cluster that is active. The waking of a cluster is essentially an inter-cluster move. An active cluster moves an instruction address to the PC register of the sleeping cluster, and tells the sleeping cluster to start execution from that PC after a certain number of cycles so that all clusters remain synchronized. A WAKE operation is added to the ISA that has three source operands: a cluster id indicating which cluster to wake up, a start PC where the sleeping cluster should begin executing, and a delay, which is the number of cycles the sleeping cluster should wait before it starts executing. This delay operand is used to give more scheduling flexibility when multiple WAKEs must be executed and the inter-cluster bandwidth is limited. The WAKE operation is also guarded by a predicate.

Figure 5 shows an example usage of sleep mode in DVLIW. Figure 5(a) is a portion of a control flow graph; edges labeled TK (NT) represent taken (fall-through) paths. Figure 5(b) shows the same control flow graph on a 2-cluster DVLIW machine. Assume the program does not have enough parallelism for the compiler to assign any operations to cluster 1 in blocks BB2, BB3, BB4, and BB5. Without sleep mode, all of these blocks must contain EPBR, EBR and NOPs to stay synchronized. With support for sleep mode, a SLEEP_TK operation is added in cluster 1 at the end of BB1, at the same cycle as the EBR operation. If the EBR is taken, cluster 1 will enter sleep mode. To wake up cluster 1, a WAKE operation is added in BB5 of cluster 0, one cycle before the branch. It will make cluster 1 wake up and start execution at BB6 after a 1-cycle delay, if the branch is not taken (p2 is false).

4.2 Compiler Support

In addition to performing the conventional VLIW compiler tasks, such as scheduling and register allocation, a DVLIW compiler must partition operations across the clusters, insert ICMOVE sequences to transfer values between clusters, insert EPBRs, BCASTs and

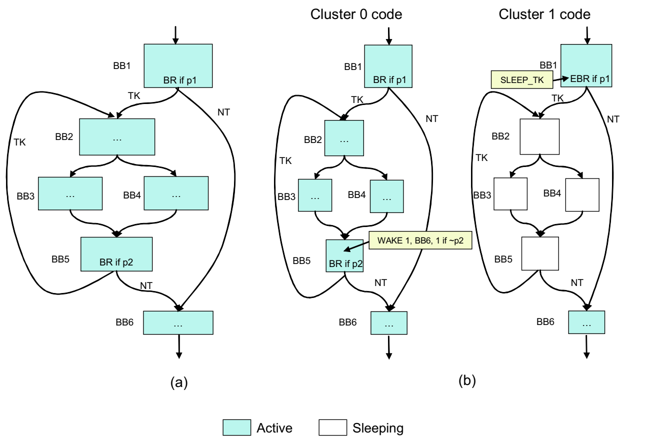

Figure 5: Sleep mode example: (a) Original control flow graph, (b) control flow graphs for a 2-cluster DVLIW where cluster one is in sleep mode for blocks two through five. Sleep mode is entered via the SLEEP_TK operation in BB1 on Cluster 1 and exited via the WAKE operation in BB5 on Cluster 0.

EBRs to orchestrate the correct control flow, and insert SLEEPs and WAKEs to switch clusters between active and sleep modes.

The first step is to perform operation partitioning to assign operations to clusters [11, 6]. We use a typical partitioning algorithm that attempts to maximize performance by balancing the application workload across the clusters while minimizing the number of required ICMOVEs. If a value must be communicated from one cluster to another, an ICMOVE must be inserted; if this ICMOVE is on the critical path, performance may suffer. In addition, the inter-cluster communication bandwidth is limited. Therefore, the goal is to localize communication of data values within a cluster as much as possible, while distributing work across clusters to take advantage of instruction- and loop-level parallelism.

This performance-centric clustering algorithm attempts to make use of all of the parallelism available in the machine in order to achieve the best schedule for each region of code. In many cases, this results in an unbalanced schedule, with more operations on some clusters than on others. For example, if there is a long dependence chain in the code, this dependence chain may be scheduled on one cluster, while other independent operations are scheduled on other clusters. In such cases, it is often possible to proactively offload all of the operations from some lightly-loaded clusters while suffering little or no performance penalty. Those clusters can then be placed in sleep mode as described in Section 4.1, saving energy and code size.

The overall compiler flow is summarized as follows. First, operation partitioning is performed, followed by ICMOVE insertion. Next, the compiler inserts EPBRs and BCASTs to the appropriate clusters for every branch sequence in the code. The code is then scheduled and register allocated. After scheduling, EBR operations are inserted to ensure all clusters contain a branch. EBRs are inserted in the same cycle as the original branches. Note that the branch resource is guaranteed to be free as at most one branch is allowed per cycle in the original schedule. Finally, the active and idle blocks in all clusters are identified and the appropriate SLEEP and WAKE modifiers on the branches are inserted.

4.3 Instruction Prefetching

In DVLIW, the centralized I-cache is distributed to several smaller caches. When a program is executed, every I-cache miss requires all clusters to stall. Thus, the smaller distributed I-caches can cause more stalls due to multiplicative misses. Many of these misses may be handled in parallel; however, differing code sizes in each cluster can cause misses not to overlap perfectly. Moreover, it may not be possible to evenly partition the code among clusters, further increasing stall cycles when each cluster has a smaller I-cache. To mitigate the penalty of I-cache misses, we employ the HPL-PD architecture mechanism for software-controlled instruction prefetching to reduce stalls [15]. The idea is that PBR and EPBR operations contain a prediction bit to initiate a prefetch of the target address of the branch. An integer field is also added to PBR/EPBR operations to specify the number of cache lines to prefetch. Note that prefetching is also useful on a traditional VLIW; however, it is more important in DVLIW machines due to the smaller distributed I-caches.

4.4 Procedure Calls

There are several issues with procedure calls in DVLIW. First, as stated in Section 4.1, sleep and wake operations are inserted at branches. If a branch is a procedure call, the compiler may not know which clusters are active in the callee function particularly if procedures are compiled separately. Moreover, since a procedure may have multiple callers, it is impossible to know which clusters are active at the return point (call site). The second issue is function pointer support. If a function is called through a pointer, its address will be determined at run time. In DVLIW, a function has multiple start addresses, one for the code in each cluster. Thus, each function pointer in DVLIW needs to be a vector of addresses instead of a single address. This solution requires non-trivial compiler changes and could significantly increase the code size if function pointers are heavily used. Finally, for functions in dynamic linked libraries (DLLs) and interrupt/exception handlers, a single address is used to represent the branch target in traditional processors. If a vector of addresses is used to represent a branch target in DVLIW, the compiler, operating system, and user software would all need to change.

Our solution to all of these problems is to enforce a DVLIW calling convention. We assume that only the first cluster (cluster 0) is active in the first and last block of every function. If the function needs to use more than one cluster, it will have explicit wake operations to activate them. In this case, every function has only one start address, just as in a traditional processor. Thus, function pointers and library calls can be handled smoothly. Note that this calling convention can be relaxed for certain function calls if the source code for both caller and callee are available, and the compiler is able to perform inter-procedural analysis to determine the start addresses and sleep/wake states for the callee function. An algorithm can be designed to intelligently choose between different ways to handle function calls to achieve optimal performance and energy usage. Currently, we assume separate compilation, so no inter-procedural optimizations are applied.

Function pointers are a kind of indirect jump where the target is another function. Indirect jumps may also target basic blocks within the same function, such as in table jumps generated for C switch statements. In this case, the compiler simply replicates the branch target table for every cluster.

5. EXPERIMENTAL RESULTS

To evaluate the DVLIW architecture, an experimental system including compiler, assembler, linker, and simulator was built using the Trimaran toolset [2]. The DineroIV trace-driven cache simulator [10] was integrated into the simulator to provide cache performance data.

2-cluster and 4-cluster DVLIW processors are compared to baseline VLIW machines with a traditional centralized control path. Both DVLIW and baseline machines have a clustered datapath. Common to all machines are 2 integer units, 1 floating-point unit, 1 memory unit, and 1 branch unit in each cluster. Operation latencies similar to those of the Intel Itanium are assumed. The L1 I-caches are 4-way with 64-byte blocks; total sizes of 8K, 16K, and 32K are examined. Each L1 I-cache in a DVLIW cluster has a size of $\frac{total\ L1\ size}{\#clusters}$. For example, in the four-cluster configuration, a DVLIW machine with four 4K L1 I-caches is compared to a machine with a conventional 16K L1 I-cache. 32K data caches are assumed for all experiments. The L1 miss latency is a minimum of 10 cycles. All machines have a unified, centralized 8-way 256K L2 cache with 128-byte blocks. The clusters are configured in a mesh for both DVLIW and the baseline. The latency for inter-cluster communication is 1 cycle per hop.

The compiler employs hyperblock region formation; loops were software-pipelined if possible and unrolled otherwise. Unless otherwise specified, sleep mode is used for DVLIW machines. The performance of the MediaBench benchmarks [19] and a subset of the SPECint2000 benchmarks were evaluated.[1] The multimedia benchmarks have characteristically high ILP, making them ideal candidates for wide-issue machines, while the SPEC benchmarks are more irregular and difficult to partition effectively.

DVLIW vs. CVLIW Cost and Energy. The experimental analysis is begun by comparing the cost and energy of the DVLIW instruction fetch and distribution subsystem to that of a traditional VLIW which uses a centralized control path architecture, CVLIW for short. Table 1 compares hardware cost and power consumption of the instruction align/distribution logic for DVLIW and CVLIW. The issue width of all configurations is 16 and the TINKER encoding is assumed [8]. To employ compressed encodings, shift, align, and distribution networks are required in both CVLIW and DVLIW. The shift/align network is required because the beginning of variable-length instructions will not always be aligned to the beginning of a cache line. A distribution network is required because the control bits must be delivered to all the FUs but the positions of control bits for FUs are not fixed. The cost of these networks increase quadratically with the issue width, because the number of MUXes and the number of inputs for each MUX increase with the issue width. Using the Synopsys Design Compiler [1], we synthesized the major components of these networks. As can be seen from the table, the distributed organization of the DVLIW can effectively control this quadratic cost and energy consumption of the alignment and distribution logic. By distributing the networks, a large centralized network can be avoided for wider issue machines.

The key benefit of DVLIW is that most instruction fetch and distribution can be done locally. Only rarely is global distribution of control bits necessary. For purposes of this analysis, we define a global signal as an inter-cluster transfer or transfer between a centralized memory and a cluster. A local signal is one that is contained within a cluster (possibly including a local I- or D-cache.) Figure 6 compares the number of bytes distributed globally on a 4-cluster DVLIW and CVLIW. Three bars are shown for each benchmark. The first bar shows the increase in inter-cluster data communication on DVLIW architectures, including BCASTs and WAKEs; this increase is generally negligible. The second bar shows the ratio of globally distributed instruction bytes on the CVLIW to globally distributed instruction bytes on DVLIW. On the CVLIW with centralized I-fetch, all the instruction bits need to be distributed on global wires (both L1 hits and misses), while on DVLIW, only L1 cache misses need to be transfered on global wires. The ra-

[1]Benchmarks in the suites but missing from the evaluations were left out due to problems with the Trimaran compiler system.

Confi guration	**16-issue Traditional VLIW**	2×8 **DVLIW**	4×4 **DVLIW**	8×2 **DVLIW**
Hardware area (mm^2)	6.317	2.638	0.843	0.307
Power (μW)	95.112	46.792	25.296	12.504

Table 1: Comparison of hardware cost and power consumption for the instruction align/shift/distribution networks of various VLIW configurations.

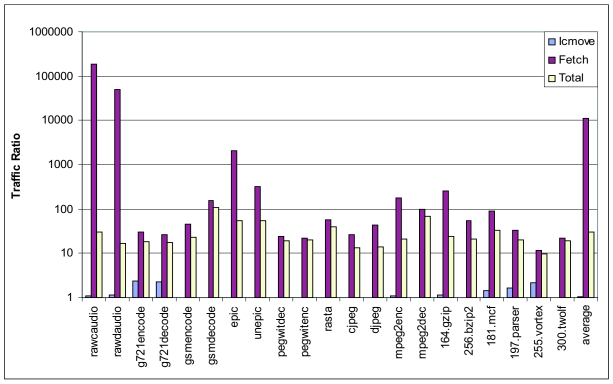

Figure 6: Reduction in dynamic global communication signals of a 4-cluster DVLIW processor.

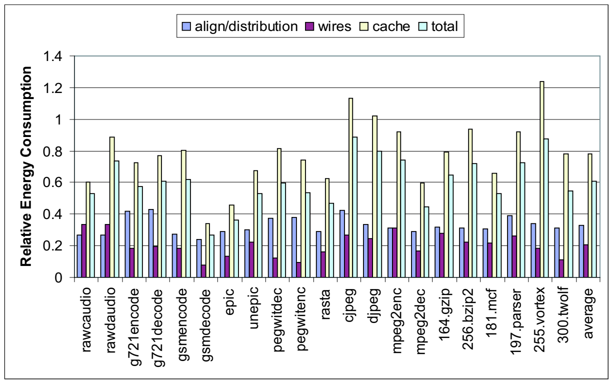

Figure 7: Relative energy consumption for the instruction fetch subsystem of a 4-cluster DVLIW processor.

tio varies from 20 to 1,000,000 across the benchmarks. The third bar shows the total reduction of global traffic including both inter-cluster move and instruction fetch. The total ratio is not as large as the fetch ratio because inter-cluster data transfers represent a significant portion of the global traffic that does not change much on DVLIW.

The reduction in global traffic combined with a more efficient cache organization leads to potential energy savings for DVLIW over a traditional VLIW. Focusing on the instruction fetch subsystem, Figure 7 reports the relative energy on a 4 cluster DVLIW architecture. A value of 1 in the figure is the instruction fetch energy consumption on a CVLIW processor with the same issue width and centralized control path. Our instruction energy consumption model consists of 3 parts: align/distribution, interconnect, and the L1/L2 caches. The align/distribution energy was presented previously in Table 1. The interconnect energy corresponds to the energy consumed to broadcast the control signals from the I-cache to the FUs. We use a wire energy model similar to that of [29]. Finally, the L1/L2 cache energy is derived from Cacti [27].

As shown in Figure 7, DVLIW on average reduces the energy consumption of the instruction align/distribution network by 67%, interconnect by 80%, and the caches by 21%. The total energy consumption on the control path is reduced by 54% on average. One thing to notice is that three benchmarks (cjpeg, djpeg and 255.vortex) consume more I-cache energy in the DVLIW processor. This is because although DVLIW has smaller and more efficient caches, the number of I-cache accesses and L1 misses for these benchmarks increases by a large amount. However, the energy savings on the instruction align/distribution network and interconnect compensate for the increase in I-cache energy consumption, and the total energy coa large fraction of control path is reduced. In a real processor, instruction fetch consumes a large fraction of power dissipated by the processor. For example, instruction consumes 27% of CPU power in the StrongARM SA-100 [9], and almost 50% of CPU power in the Motorola MCORE [20]. The energy savings on the control path translate to a significant power reduction for the whole chip.

DVLIW vs. CVLIW Performance. Figure 8 shows the percentage change in execution time on 4-cluster DVLIW versus a 4-cluster CVLIW. Two bars are presented for each benchmark; they represent total I-cache configurations of 16K and 32K. The baseline for each bar is the execution time on the CVLIW machine with the corresponding total I-cache size. A positive value means the execution time was longer on DVLIW. Each bar in the graph is divided into 2 parts; the lower part represents the portion of execution time due to the increase in computation, such as the EPBRs, BCASTs and WAKEs. The upper part represents the change in I-cache miss stalls.

On average, the execution time increases due to the extra computation is about 5%. For gsmencode and gsmdecode, the computation time is shorter because a different clustering algorithm is used for DVLIW to set more clusters to sleep mode, which could result in a different schedule. A closer look into the data shows that most of the slowdown in the computation is because some BCASTs are inserted on the critical path and thus increase the schedule length. This observation suggests that instruction replication techniques [5] can be used to reduce execution time on DVLIW in the future.

The I-cache stalls vary widely across different benchmarks and I-cache configurations. The L1 cache misses increase in the DVLIW architecture for most benchmarks because the code size on DVLIW increases, and the code is not evenly distributed across clusters while the I-cache hardware is evenly divided between the clusters. A more detailed study of the g721encode benchmark on the 16K I-cache configuration shows that a large unrolled loop body can fit into the 16K centralized I-cache, but in the 4-cluster DVLIW, the I-cache for each cluster is only 4K. The loop is not divided evenly and cannot fit in the 4K I-cache in one of the clusters. This results in cache misses every iteration of the loop.

To better illustrate the overhead of managing the separate instruction streams on DVLIW, DVLIW is compared against CVLIW. One thing to note is that the CVLIW machine is not scalable. As the number of FUs increases (up to 20 in these experiments), it becomes unrealistic to implement a VLIW with centralized control. Thus, DVLIW is compared with an ideally-scaled CVLIW instead of a realistic design in the prior experiments. The modest overhead of DVLIW should therefore be considered a positive result.

DVLIW Sleep Mode. To demonstrate the effectiveness of sleep mode in reducing code size, Figure 9 compares the relative code size with and without sleep mode enabled for a 4-cluster DVLIW machine. The code size is reported relative to the 4-cluster CVLIW with a fully compressed encoding. On average, DVLIW code size

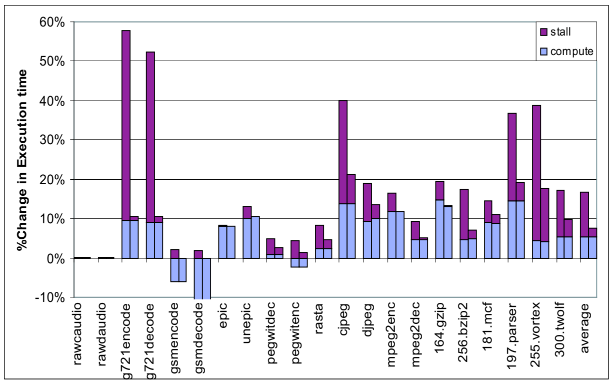

Figure 8: Relative execution time on a 4-cluster DVLIW processor with total I-cache sizes of 16K and 32K.

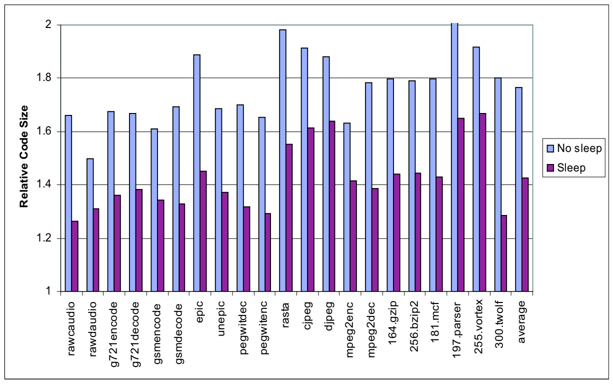

Figure 9: Effectiveness of sleep mode on a 4-cluster DVLIW.

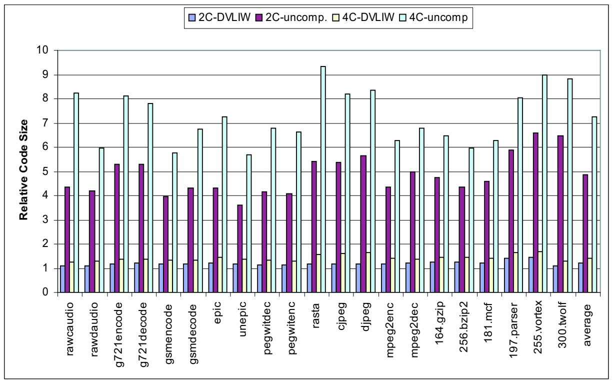

Figure 10: Comparison of the relative code size of 2- and 4-cluster DVLIWs with VLIWs having a distributed I-cache and a shared PC.

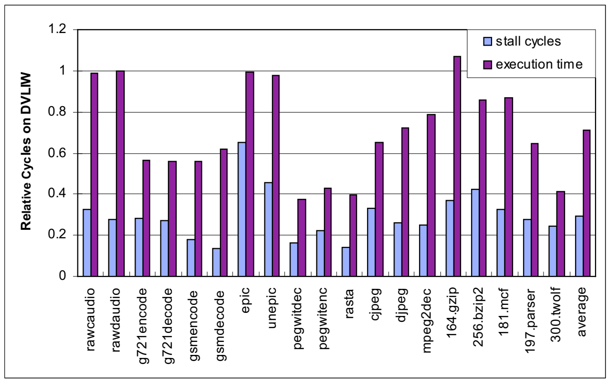

Figure 11: Comparison of the relative stall cycles and execution time on a 4-cluster DVLIW with a VLIW having a distributed I-cache and a shared PC.

decreases from 1.77 times that of a CVLIW without sleep mode to 1.40 when sleep mode is used. This shows that in parts of the application that do not have high ILP, significant code size and energy savings can be realized by turning off unused clusters.

DVLIW vs. Distributed Uncompressed Instruction Memory. The obvious alternative to DVLIW is to distribute the control path, but with a single PC, similar to the Multiflow TRACE system as shown in Figure 1(c). With this approach, code compression cannot be employed as discussed in Section 2. Figure 10 compares the normalized code size of DVLIW and the distributed uncompressed architecture. On the y-axis, a value of 1 is the code size of the benchmark on a CVLIW machine with a fully compressed encoding. Again, the TINKER encoding is assumed in our experiments [8]. Four bars are presented for each benchmark. The first bar is the normalized code size of the benchmark on the 2-cluster DVLIW machine. The second bar shows the normalized code size on a 2-cluster distributed machine with uncompressed encoding. The third and fourth bars show corresponding data for 4-cluster configurations. On average, the normalized code size is 1.2 on the 2-cluster DVLIW and 1.4 on the 4-cluster DVLIW. The increase in code size on DVLIW is due to the introduction of EPBRs, BCASTs, EBRs, WAKEs and other related operations such as spill code. As stated before, the fully uncompressed encoding must be used for distributed architectures with a shared PC or multiple identical PCs. As shown in the figure, the code size on DVLIW machines is much smaller than the code size on those machines.

Figure 11 shows the relative stalls and execution cycles of a 4-cluster DVLIW over the corresponding processors with uncompressed distributed I-cache. The datapath of both machines is identical. The TINKER encoding is used for DVLIW, while the shared PC machine uses an uncompressed encoding. As shown in the figure, on average, stall cycles decrease by 70% on DVLIW and consequently, the total execution time reduced by 30%. One benchmark 164.gzip is slightly slower on DVLIW because the I-cache stalls account for a small fraction of the total execution time, and the overhead incurred by extra operations on DVLIW outweighs the benefit of fewer I-cache stalls. However, for the majority of benchmarks, the benefits of smaller code size and fewer I-cache misses more than compensate for the overhead of extra operations.

6. CONCLUSION AND FUTURE WORK

In this paper, we have proposed a distributed VLIW architecture to overcome the scalability limitations of VLIW control paths. The architecture combines a multicluster datapath design and a completely decentralized instruction fetch, decode, and distribution subsystem. The DVLIW architecture simplifies the dispersal of complex VLIW instructions while not sacrificing the code size benefits of compressed instruction encodings. DVLIW executes multiple instruction streams at the same time, orchestrated to collectively function as a single logical stream through architectural support and compiler technology. Experimental results show that on average DVLIW reduces inter-cluster communication traffic by 30x. This leads to a 90% savings in interconnect energy and a 21% overall energy savings in the instruction fetch subsystem. DVLIW introduces a small performance overhead over ideally scaled VLIW machines with centralized control paths due to increased instruction cache stalls, ranging from an average of 17% with an 16k instruction cache to 7% with a 32k instruction cache. DVLIW also introduces a modest code size increase due to the explicit management of the multiple instruction streams. However, the code size on DVLIW is still about 5 times smaller than the code size on other distributed architectures with a shared PC or multiple identical PCs where uncompressed encodings must be used. We believe DVLIW provides a highly scalable architectural model where processor execution width can be efficiently scaled to meet the high-performance demands of current and future embedded processing.

For future work, the DVLIW architecture can be extended to support decoupled execution. In decoupled mode, every cluster executes and stalls independently. The only places that clusters must synchronize are inter-cluster register or memory communication points. To handle both, a light-weight software-controlled synchronization mechanism can be added to the baseline DVLIW. The decoupled execution mode allows the processor to better hide memory latencies and provide high performance.

7. ACKNOWLEDGMENTS

We thank Mike Chu, Nate Clark, Manjunath Kudlur, Steve Lieberman, Hyunchul Park, and Rajiv Ravindran for their help with Trimaran and willingness to answer countless questions. We also thank the anonymous referees for their excellent suggestions and feedback. This research was supported by the National Science Foundation grants CCR-0325898 and CCF-0347411, and equipment donated by Hewlett-Packard and Intel Corporation.

8. REFERENCES

[1] *Synopsys Design Compiler*. http://www.synopsys.com.

[2] *The Trimaran Compiler Infrastructure for Instruction-Level Parallelism*. http://www.trimaran.org.

[3] S. Aditya, S. Mahlke, and B. Rau. Code size minimization and retargetable assembly for custom EPIC and VLIW instruction formats. *ACM TODAES*, 5(4):752–773, 2000.

[4] V. Agarwal et al. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proc. 27th Intl. Symposium on Computer Architecture*, pages 248–259, June 2000.

[5] A. Aletá, J. M. Codina, A. González, and D. Kaeli. Instruction replication for clustered microarchitectures. In *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 326, 2003.

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

[7] R. Colwell et al. Architecture and implementation of a VLIW supercomputer. In *Proc. 1990 Conference on Supercomputing*, pages 910–919. IEEE Computer Society Press, 1990.

[8] T. Conte et al. Instruction fetch mechanisms for VLIW architectures with compressed encodings. In *Proc. 29th Intl. Symposium on Microarchitecture*, pages 201–211, Dec. 1996.

[9] D. Dobberpuhl. The design of a high-performance low-power microprocessor. In *Proc. of ISLPED*, pages 11–16, 1996.

[10] J. Elder and M. Hill. Dinero IV trace-driven uniprocessor cache simulator.

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

[12] J. Fisher. Very long instruction word architectures and the ELI-52. In *Proc. 10th Annual International Symposium on Computer Architecture*, pages 140–150, June 13–17, 1983.

[13] E. Gibert, J. Sanchez, and A. Gonzalez. An interleaved cache clustered VLIW processor. In *Proc. 16th Intl. Conference on Supercomputing*, pages 210–219, June 2002.

[14] E. Gibert, J. Sanchez, and A. Gonzalez. Flexible compiler-managed L0 buffers for clustered VLIW processors. In *Proc. 36th Intl. Symposium on Microarchitecture*, pages 315–325, Dec. 2003.

[15] V. Kathail, M. Schlansker, and B. Rau. HPL-PD architecture specifi cation: Version 1.1. Technical Report HPL-93-80, Hewlett-Packard Laboratories, Feb 2000.

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

[17] M. Kozuch and A. Wolfe. Compression of embedded system programs. In *Proc. 1994 International Conference on Computer Design*, pages 270–277, 1994.

[18] S. Y. Larin and T. M. Conte. Compiler-driven cached code compression schemes for embedded ILP processors. In *Proc. 32nd Intl. Symposium on Microarchitecture*, pages 82–92, Dec. 1999.

[19] C. Lee et al. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. 30th Intl. Symposium on Microarchitecture*, Dec. 1997.

[20] L. Lee et al. Low-Cost Embedded Program Loop Caching - Revisited. Technical Report CSE-TR-411-99, Univ. of Michigan, 1999.

[21] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang. Instruction selection using binate covering for code size optimization. In *International Conference on Computer-Aided Design*, pages 393–399, 1995.

[22] D. Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, 30(9):37–39, 1997.

[23] C. Newburn, A. Huang, and J. Shen. Balancing fi ne- and medium-grained parallelism in scheduling loops for the XIMD architecture. In *Proc. PACT-1993*, pages 39–52, 1993.

[24] B. R. Rau, W. Y. D. W. L. Yen, and R. A. Towle. The cydra 5 departmental supercomputer. In *IEEE Computer*, volume 22, pages 12–35, 1989.

[25] J. Sánchez and A. González. Modulo scheduling for a fully-distributed clustered VLIW architecture. In *Proc. 33rd Intl. Symposium on Microarchitecture*, pages 124–133, 2000.

[26] K. Sankaralingam et al. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proc. 30th Intl. Symposium on Computer Architecture*, pages 422–433, June 2003.

[27] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An integrated cache timing, power, and area model. Technical Report WRL-2001-2, Western Research Laboratory, Feb. 2001.

[28] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. 22nd Intl. Symposium on Computer Architecture*, pages 414–425. ACM Press, 1995.

[29] C. N. Taylor, S. Dey, and Y. Zhao. Modeling and minimization of interconnect energy dissipation in nanometer technologies. In *Proceedings of the 38th conference on Design automation*, pages 754–757. ACM Press, 2001.

[30] M. Taylor et al. Sclar operand networks: On-chip interconnect for ILP in partitioned architectures. In *Proc. 9th Intl. Symposium on High-Performance Computer Architecture*, pages 341–343, Feb. 2003.

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

[32] H. Wang, L. Peh, and S. Malik. Power-driven design of router microarchitectures in on-chip networks. In *Proc. 36th Intl. Symposium on Microarchitecture*, pages 105–114, Dec. 2003.

[33] A. Wolfe and J. P. Shen. A variable instruction stream extension to the VLIW architecture. In *Proc. ASPLOS-IV*, pages 2–14, 1991.

[34] Y. Xie, W. Wolf, and H. Lekatsas. A code decompression architecture for VLIW processors. In *Proc. 34th Intl. Symposium on Microarchitecture*, pages 66–75, 2001.