Alignment Analysis for Scalable Clustered VLIW Processors

March 15, 2002

Abstract

Increases in global wire delays present substantial challenges for conventional microprocessor designs. Latencies of large, centralized structures limit either cycle time or IPC, making it difficult to provide performance improvements. A promising strategy consists of clusters or tiles of processing units with local control and storage. Clustering can substantially reduce global communication, particularly if most memory accesses can be satisfied in a local level-1 data cache. Compared with a monolithic design, smaller independent memory banks operate with low latency and can be accessed in parallel to provide high bandwidth.

In this paper, we explore the design space for fully-clustered VLIW architectures. We describe and examine trade-offs in cluster control synchronization, inter-cluster communication, and partitioned caching. Using detailed area and wire-delay models, we estimate the intercluster communication and memory latency of several different floorplans and caching strategies. To achieve the best performance with clustered memory systems, we have developed a suite of compiler transformations and analyses that determine the bank location of most memory references statically. These techniques reduce non-local memory accesses by a factor of 2.5 and result in an overall performance improvement of 33%.

1 Introduction

The first VLIW processors [10, 6] were designed to span multiple boards, which necessitated clustering at the architectural level. Although these machines did not contain caches, each cluster could quickly access a single bank of a partitioned memory. Architects realized early on [9] that intelligent compilation was crucial to the success of these designs. However, compiler technology for clustered VLIWs was still in its infancy. Over the next few decades, advances in process technology allowed designers to build complex processors (including VLIWs) on a single chip, thus obviating the need for clustering.

Technology trends are now changing this equation, as faster transistors and slower wires are increasing the cost of global on-chip communication. Projections show that even pedestrian clock rates will allow only 1.4% of a $20mm \times 20mm$ chip to be reached in a single clock cycle [2]. These trends have caused recently designed superscalar machines, such as the Alpha 21264 [15] to partition the microarchitecture into clusters with separate register files. Thus far, such machines still subscribe to a unified memory design.

When wire delays limit the speed of global communication, there are many constraints imposed on a traditional VLIW design. First, the global implicit synchronization required by the lock-step execution model may not be feasible for wide-issue VLIWs. For high-speed designs, branch conditions and stall signals cannot be communicated across all clusters in a timely manner. Second, monolithic and centrally located shared caches may not provide sufficiently high bandwidth or low latency access to memory. In this paper, we briefly describe the issues and potential solutions for global VLIW control and examine memory system designs for clustered VLIW architectures in more detail.

We propose the use of a banked level-1 cache in which each bank is located near one cluster to provide high speed local memory access. Addresses are mapped across the banks so that consecutive memory elements are low-order interleaved across the clusters. Each cluster gains fast access to a *local* memory bank in exchange for longer latency to remote banks. Low-order interleaving allows us to take advantage of fine-grain data parallelism, if instructions can be mapped to the clusters where the needed data is located.

Novel compiler techniques are needed to obtain good performance on such fully clustered architectures. This paper describes how *Alignment Analysis* can be used to reduce the communication overheads of a distributed memory system. When used in conjunction with a suite of program transformations, our compile-time analysis drastically reduces the number of non-local memory accesses. This strategy effectively negates the impact of slower intercluster memory latency.

With near-term (100nm) technology specifications and aggressive wire delay models, the difference between local and remote memory access is only a factor of two. Consequently, the overall performance benefits of optimizing for local accesses in a clustered design were less than we initially expected. However, these results are sensitive to the ratio of remote to local latencies. When the ratio was increased to a factor of four, alignment analysis provided a speedup of more than 65%. However, we discovered a surprising secondary effect of our compiler analysis. By computing a schedule that balances memory operations across clusters, we can substantially increase overall performance of a VLIW architecture, regardless of the memory model. In fact, this algorithm performs better than a state-of-the-art instruction scheduler.

The remainder of this paper is organized as follows: The following section details the tradeoffs of clustered VLIW design. In Section 3 we discuss our compiler analyses and transformations that align memory

operations to specific banks. We explain our partitioning and instruction scheduling algorithms in Section 4. Section 5 describes experimental methodology. Section 6 presents our simulations results. Finally, we outline related work and conclude.

2 Clustered VLIW Design Tradeoffs

Most of the performance improvements in modern computer systems have come from clock rate growth, which itself is a combination of advancements in process technology, and deeper pipelines that reduce the amount of work per cycle. Unfortunately, deepening pipelines has only a limited future; research suggests that pipelining itself could at best only provide another factor of two in performance [13]. Consequently, future performance increases must stem from exploiting greater degrees of concurrency within the instruction stream. Adding execution units and the logic to control them is made possible by the exponential growth in transistors that can be fabricated on a single silicon substrate. While device scaling will continue into the future with sub-100nm transistors numbering in the hundreds of millions per chip, on-chip wiring will not keep pace [2]. Increasing communication latencies are already affecting modern-day microprocessors; two of the twenty pipeline stages of the Pentium 4 are dedicated to data transmission across long wires [12]. These wiring delays present substantial challenges to conventional centralized architectures that rely on instantaneous global communication.

The alternative sought by recent research and development is clustering, which partitions the architecture, attempting to localize communication. Clustering also takes global communication off of the critical path. For example, the Alpha 21264 consists of two distinct clusters, each with its own execution units and register file [15]. Many papers have been published on the design of distributed issue windows and reorder buffers for clustered superscalar architectures. While modern superscalars can successfully detect instruction dependencies at runtime, dynamically partitioning instructions so as to minimize memory latencies is far more challenging. Thus, clustered architectures will likely rely on compiler analysis.

Very Long Instruction Word (VLIW) architectures have long been proposed as machines that can efficiently exploit instruction level parallelism. More recently they have been billed as a less complex alternative

to dynamically scheduled out-of-order processors. However, these systems are also subject to the the VLSI scaling trends of more transistors, but slower wires. Large common structures such as register files and shared caches cannot be made larger or augmented with additional ports without substantially increasing size and access time. Global communication paths, such as bypass paths and instruction sequencing control communication will also be slow, particularly if the communication must cross a substantial distance within the chip.

In the following sections we examine the design issues in highly clustered VLIW processors, in which register files and data cache banks are partitioned across clusters. We explore how physical topology and placement of clusters and their memories can affect overall performance. The details of the topology and its impact on latencies are discussed in detail in Section **??**. In the remainder of this section we first discuss the alternatives in level-1 cache design for a clustered VLIW processor. We then describe the design constraints and alternatives for VLIW data communication and instruction sequencing that give the illusion of lock-step instruction execution without placing cross-cluster synchronization on the critical path.

2.1 Memory Arrangement

An important consideration in the design of a clustered VLIW is the structure and layout of level-1 cache. Traditional designs treat memory as a monolithic structure, with uniform latency to all clusters. However, subdividing a cache into multiple banks can provide several benefits to wider issue processors. First, multiple banks can simultaneously provide service to several memory requests. In the absence of bank conflicts, memory bandwidth increases. Second, access latency of a memory array is a function of its capacity: a smaller bank can be accessed faster than a larger one. Finally, a bank can be placed close to the source of memory requests, reducing transmission latency and accelerating accesses. There are two basic techniques for mapping and managing the data in a partitioned cache.

Cache Coherence: One approach to clustered memory design is to share data among banks using a cache coherence protocol. This scheme is often used in multiprocessor environments where cache memory is separated among processing elements. Multiprocessors are well-suited to exploit coarse-grain parallelism.

Cache coherence works well when each processor operates on distinct data. In a uniprocessor design, however, we expect to exploit fine-grain parallelism. In this situation, the drawbacks of cache coherence can have a more pronounced effect. For example, memory space is not used efficiently if multiple copies of data are maintained. Furthermore, if ILP compilation is successful, we would expect different clusters to operate on data that have close spatial locality. In this case, false sharing could cause a serious performance degradation.

Address Interleaving: An architecturally simpler alternative to coherence is to use low-order interleaving to divide memory evenly among the banks. Low-order interleaving makes efficient use of available on-chip memory and allows us to take advantage of fine-grain data parallelism. Because of these advantages, this scheme was used in early vector machines and VLIW supercomputers. Since these machines did not contain caches, memory design was straightforward. In a design with interleaved caches, it is interesting to examine different replacement policies. In this paper, we study two alternatives.

With a *shared-tag* organization, all banks of the cache are managed cooperatively and each cache line is striped across the banks. For performance reasons, each bank may have its own physical tag array, but all tag arrays hold identical mappings. When a miss occurs in one bank, a new line is fetched and spread across all banks, replacing the same location in each bank.

An *independent-tag* second policy allows the cache banks to be managed completely independently. The addresses employed by the processor are still mapped across the banks, but a cache line is composed of non-contiguous addresses on a fixed stride; the stride is equal to the number of cache banks. On a cache miss, a new non-contiguous line is fetched into the bank that caused the miss, while leaving the other banks undisturbed. Note that through some simple address bit remapping, the non-contiguous words of a cache line can actually be contiguous in the level-2 cache and main memory. This scheme is simpler than the previous one which interleaves each cache line across the banks as no inter-bank communication is required to replace a line.

These memory designs will only provide performance gains if most memory references access local banks. Otherwise, communication of these data to other clusters will nullify the gains achieved from clustering. The selection of mapping policies depends on the spatial locality that can be exposed by the compiler. The

shared-tag method spreads the locality across the clusters, while the independent-tag policy concentrates the spatial locality within a cluster.

In section 3 we describe compiler techniques that allow us to isolate static memory references to a single bank and we examine empirically the respective benefits of the shared-tag and independent-tag organizations. For either banked-memory design, this compiler technology is key to achieving low latency and high bandwidth memory access, as well as high program performance.

2.2 Inter-cluster Communication

Beyond reducing latencies between clusters and memory banks, a successful VLIW must mitigate the latencies between the clusters themselves. One source of performance degradation is register communication latency between clusters. Reducing intercluster data communication latency is desirable as it enables dependent instructions to be placed more easily on separate clusters. This improves the load balance and instruction level parallelism of VLIW programs. Various interconnection strategies have been explored for intercluster communication, including busses and crossbar switches. As cluster count and clock speed increases, fast communication must be constructed from point-to-point links rather than slow, shared busses [7]. Parcerisa et al. proposed and evaluated dynamically routed and switched point-to-point networks for intercluster communication in a clustered superscalar processor [18]. While a similar strategy could be used for VLIWs, contention for any limited communication paths or remote bypass ports can be eliminated through careful scheduling.

A second source of intercluster latencies stem from control signals required to preserve lock-step execution across all of the clusters. Sequential execution of instructions on each cluster is not difficult, but the challenge arises when the clusters must all branch at the same time, or must all stall due to an unexpected long latency memory access. If the latency to synchronize all the clusters on every cycle is significant, then the constant synchronization and corresponding stalls will devastate performance. One method to mitigate the synchronization requirements between clusters is to decouple them and allow limited slippage in their schedules [14]. This can be further augmented by providing each cluster with its own program counter. All branching would be performed locally, with synchronization enforced on data arrival. A second alternative

which better preserves the VLIW model is for each cluster to predict sequential execution and for local rollback to occur when a new program counter is delivered by the cluster performing the branch. While we believe this rollback approach for branching in distributed VLIW architectures is novel, we will perform further analysis in future work.

2.3 Summary

While previous VLIW processors have been able to ignore layout driven communication constraints, future high frequency systems must address on-chip communication delays during design and evaluation. The critical paths include memory latency and cluster interaction latencies, with both inter-cluster data and control communication being important. In this paper, we examine a cluster-centric design which reduces intercluster latencies at the expense of local memory latencies. We then focus on the methods for mitigating the remote cache access latencies in a clustered VLIW architecture.

3 Compiler Description

In multi-bank designs, consecutive words are typically low-order interleaved across the banks. Such a scheme allows us to take advantage of fine-grain memory parallelism. When a static memory reference accesses the same bank for each dynamic instance, we say that the references is *aligned*. Aligned memory operations offer a large potential for improving performance on a fully-clustered VLIW. If we can ascertain the bank-alignment of the reference at compile-time, we can schedule computation that access these data on the local cluster, improving memory latency. Furthermore, accesses to different banks can be safely executed in parallel.

To achieve this end, we have developed a dataflow analysis that computes the alignment of static memory references [16]. When aligned references exist within the program, our algorithm is very successful at detecting them. However, very few references are actually aligned in practice. For a four bank machine, only about 14% of dynamic memory references are actually aligned. The primary reason for this is illustrated in the simple loop of Figure 1(a). Here, the array reference within the loop writes consecutive memory locations

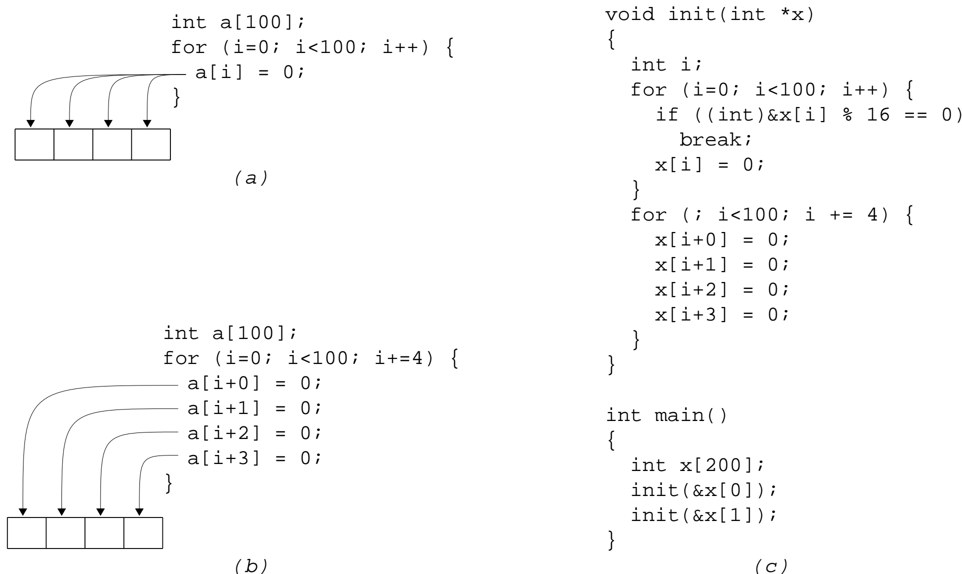

Figure 1: (a) In a machine with four memory banks, a simple loop accesses a different bank on each iteration. (b) For local and global arrays, loop unrolling ensures that each memory reference accesses a single bank. (For simplicity, the illustration assumes the iteration count is a multiple of the unroll factor.) (c) A pre-loop is used when an array parameter is passed with different bank alignments for different procedure invocations.

on each iteration.

To improve this situation, we have developed a suite of compiler transformations that strive to increase the number of aligned references. We have taken great care to ensure that all our algorithms are practical for real-world applications. Each transformation is applied locally, ensuring its scalability to large program sizes. We feel this freedom from whole-program analysis is essential to the general acceptance of any compilation system.

Alignment Conventions

To improve alignment when accessing aggregate arrays and structures, we require that these data are always placed at the same alignment. This requires that we allocate stack frames in blocks that are a multiple of the total bank width . Alignment of stack-allocated data can depend on the current position of the stack pointer. Aligning stack frames ensures that these data are aligned identically for any invocation of the enclosing procedure.

Alignment of stack frames has the added benefit that scalar stack access are always aligned to the same bank. Since the compiler is responsible for placement of data within the stack, this alignment is easily

obtained at compile-time.

Loop Unrolling

Loop unrolling is used to increase the number of aligned memory references within loop bodies. This is shown in Figure 1(b) for a machine with four memory banks. After unrolling, each memory reference only accesses a single bank. In more complex scenarios, multiple nested loops can be used to range over the elements of a multi-dimensional array. In this situation, it may be the case that the inner loop does not iterate over the low-order elements of an array. Here, simple inner-loop unrolling will not create aligned references. Loop interchange [4] or some form of outer-loop unrolling can usually correct this difficulty.

Pre-loop

Unrolling works well for accesses to local or global arrays because the base of the array is always placed at a known alignment. However, programming languages such as C allow passing of arbitrary pointers to the middle of an array. Since an array can be passed with different bank alignments for different invocations of the enclosing procedure, unrolling alone cannot guarantee single-bank accesses. To remedy this situation, we use a pre-loop as shown in Figure 1(c). The pre-loop is used to execute a few iterations of the original loop body until the memory reference within the loop reaches a known alignment. At this point, we exit the pre-loop and begin execution of the unrolled loop. As before, the references within the unrolled body are guaranteed to make single-bank accesses. As long as the majority of iterations take place in the unrolled loop, most dynamic memory references are aligned.

The loop in Figure 1 is a simple example with a single memory reference. In practice, most loops contain multiple memory operations that reference many different arrays. In this case, the break condition should contain multiple checks if we wish to guarantee bank alignment for more than one reference. In these situations, pre-loop construction can be complicated. A poor choice for the exit condition can cause all iterations to take place in the pre-loop. Details on the intricacies of pre-loop construction can be found in [16]. In short, our solution is to use profiling to observe the run-time alignment for a particular data set. Based on the profile, an exit condition is chosen to maximize the number of dynamic bank-aligned memory

operations.

Traditional profile-based systems can suffer from the fact that program transformations are based on the results of a single data set. If program characteristics vary widely with input data, profiling will not produce good results. However, we have found that bank alignment is highly immune to the particular choice of profile data set. In fact, we achieve nearly identical results regardless of input data.

4 Instruction Partitioning and Scheduling

A good instruction scheduler is critical to achieving good performance in any VLIW architecture. In clustered designs, there is the added complextity of instruction partitioning. Wrong partitioning decisions have a serious impact on performance since data will be communicated between clusters more frequently. At some point, communication costs can nullify the benefits of parallel execution.

Instruction partitioning and scheduling for a clustered VLIWs is still a very active area of research. To test our architectural designs, we have implemented two algorithms. The first is based on a state-of-the-art partitioning algorithm developed at HP Labs. The second leverages alignment information to minimize load and store latency.

4.1 Desoli's Partitioning Algorithm

Our first algorithm leverages the work done by [8]. For every instruction in a basic block, the algorithm determines the *critical path length* (CPL) from the instruction to the end of the basic block. The CPL is used to determine *sub-traces*. Sub-traces are critical sequences of instructions linked by data dependences that we attempt to assign to the same cluster. They are built using the *Partial_components* algorithm described in [8].

Next, sub-traces are mapped to clusters using a simple heuristic that tries to balance workload. In our current implementation, we move traces among clusters without taking into account the cost of eventual communication. This is done to enforce utilization of all clusters. After determining the assignment of instructions to clusters, we add inter-cluster communication where necessary and we perform list scheduling

using the CPL to prioritize instructions.

4.2 Alignment-Based Partitioning Algorithm

In order to take advantage of memory alignment, we've implemented a second partitioning algorithm that partitions instructions based on their iteraction with memory operations. The impetus for the algorithm is based on three general observations of program behavior:

- If a memory references is aligned to a single bank, the reference should be executed on that local cluster.
- After unrolling, memory references are evenly distributed among memory clusters.
- Memory references are evenly distributed throughout instruction trees, which each tree having a few references.

Based on these generalizations, our algorithm begins by mapping memory operations with known alignment to the cluster associated with the known memory bank. Instructions that produce or consume data of mapped instructions are placed on the same cluster. This process continues until all instructions have been mapped. If an instruction uses data from different clusters, it is mapped adjacent to the instruction that is closer in terms of execution order. In order to achieve good load balancing, instruction partitioning is done in a breadth-first manner.

Once instructions have been assigned to a cluster, we perform list scheduling within each cluster. Intercluster communication is inserted where necessary. If parallelism is available and alignment detection is successful, this simple heuristic reduces intercluster communication, provides good cluster utilization, and minimizes memory access times.

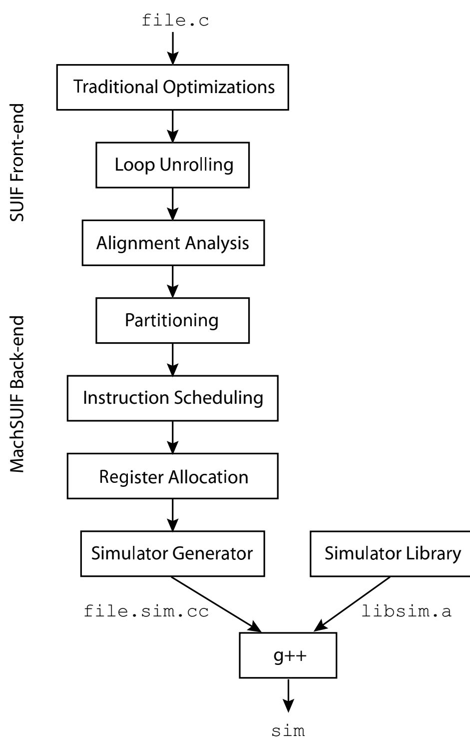

Figure 2: Compiler Flow

5 Experimental Methodology

5.1 Compiler Flow and Simulation Environment

This section describes the methodology used to collect simulation results. Figure 2 depicts our compiler tool chain. The compilation process begins in the SUIF front-end [21]. In addition to performing alignment analysis, the front-end carries out traditional optimizations such as loop unrolling, constant propagation, copy propagation, and dead code elimination.

Our VLIW back-end follows [17]. Written using MachSUIF [1], the back-end allows us to easily vary the number of clusters, functional units, and registers in the target architecture. Instruction latencies and inter-cluster communication latencies are also configurable. The partitioner and instruction scheduler use such information, combined with alignment data, to generate effective code. Similarly, our register allocator must know the number of registers in each cluster.

The result of the compilation process is a compiled simulator that we use to collect performance numbers.

The simulator accurately models the latency of each functional unit. We assume that all functional units are fully pipelined. Furthermore, the simulator enforces lock-step execution. Thus, if a memory instruction misses in the cache, all clusters will stall. The memory system is run-time configurable so we can easily isolate the performance of various memory topologies. In total, the back-end comprises nine compiler passes and a simulation library.

5.2 Chip Organization

Our design is aimed at a 15 mm X 15 mm chip at a technology of 100nm. A substantial portion of the chip, roughly 66%, is occupied by the L2 cache and the remaining area is occupied by the clusters and the L1 memory modules. The L2 Cache size is 4MB and is 8 way associative. It is internally banked and has an access time of 16 cycles at 100nm and a clock rate of 8 F04. The L2 Cache is constant in size and access time across all organizations of the clusters and the L1 cache banks. Our focus is to evaluate the impact on performance of the different organizations of the L1 cache banks and the clusters. To be able to quantitatively compare any two organizations we need to be able to calculate the latencies associated with each unit and the communication latencies between different units. In the following section we describe in detail the different cluster and memory organizations and the methodology we use for area and delay estimation.

5.2.1 Cluster Composition

In our model each cluster is composed of 2 Integer ALUs and 1 Floating point functional unit. It also has a six-ported Integer register file, to support the 2 Integer ALUs, with 64 registers each 64bits wide. The access time of the register file is 2 cycles. The Floating point register file has 64 registers each 64 bits wide and is three ported. The access time is the same 2 cycles. The cluster also contains 64 predicate registers which are each one bit wide and so occupy negligible area. Each cluster is capable of both integer and floating point arithmetic operations and the latencies associated with each operation are given in Table 1. The LD and the XFR are two instructions whose latencies depend on the relative placement of the clusters and the memory modules and we will discuss those when we discuss specific organizations. The L1 cache is always

organized as 4 banks and could either be centralized or distributed among the 4 clusters. Each L1 cache bank is 64KB and is direct mapped. So the overall L1 capacity is 256KB. Each bank has two ports and can be thought of as one local and one remote port. The access latency of one bank is 3cycles.

Instruction	Latency
ALU	1
CMP	1
MUL	5
DIV	20
MOD	20
Integer to FP conversion	4
FP add/sub/compare	4
FP multiply	4
FP divide	18
Loads	Variable
XFR (inter-cluster communication)	Variable

Table 1: Instruction Latencies

Single cluster and Centralized L1: The processor contains a single cluster, a monolithic L1 cache and an L2 cache. The cluster has 3 functional units and an Integer and Floating point register file. This organization is illustrated in Figure 3a.

Multiple clusters and Centralized L1: The processor contains a multiple clusters, a monolithic L1 cache that is internally banked and an L2 cache. Each cluster has a certain number of functional units and a register file. This organization is illustrated in Figure 3b.

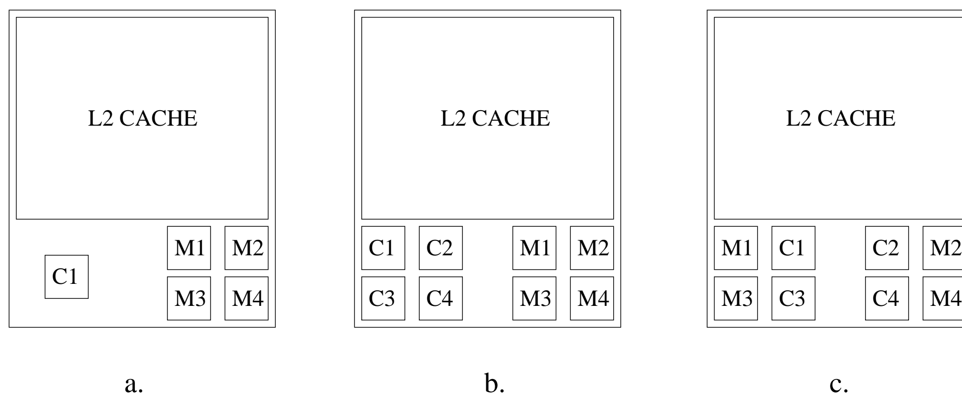

Figure 3: Multiple Clusters with a Centralized L1 and a Distributed L1

In both the organizations discussed above the L1 cache banks are organized as one monolithic memory unit that is internally banked. The access latency of the L1 cache is the maximum of the bank access

latencies. All clusters see the same worst case access latency to the L1 cache. Since the L1 is centralized the total cache access time is the bank access time and the routing delay to the edge of the cache and then to the clusters which we calculated to be 5 cycles.

Multiple clusters and Distributed L1: The processor is made up of multiple clusters and an L2 cache. All clusters are identical in the type and number of resources they have. The L1 cache banks are distributed among the clusters so that each cluster has a local L1 cache bank but also has the capability to access the other remote banks. This organization is illustrated in Figure 3c.

The architecture includes separate networks for remote memory accesses and inter-cluster communication. Each cluster has a private channel that connects to all other clusters. Similarly each L1 cache bank has a private channel between it and all the clusters that can access it. These channels are optimized for speed by using optimal repeater placement along the wires.

5.3 Delay and Area Estimation

For any organization of the clusters and the memory modules we have to estimate both the structural access latencies and the communication latencies between the remote units. To compute the communication latency between them we need to know the manhattan distance between the two and for that we need the capability of estimating the area of each cluster and L1 cache bank.

To estimate the area of the Integer and Floating point functional units we used the area model developed by Gupta *et al.* [11] which gives technology independent area estimates of these different structures. Cacti 3.0 [19] is a tool that provides both the area and the delay associated with memory structures such as caches and register files. The area of the cluster can be estimated given its composition and the individual areas of the building blocks. The latency to route a value from one unit to another depends on the placement of the individual units and their area. So for a given placement we used the area estimates we obtained from the area model and Cacti along with the wire delay model developed by Agarwal *et al* [3] to calculate the communication latency. The access latencies of the memory structures were also estimated using Cacti. Our design is aimed to run at 8F04 which is approximately 3.5 GHz at 100nm technology.

5.4 Machine Configurations

We have two axes of control, the cluster models and the memory models. The memory models are a combination of the L1 cache bank organization schemes and the memory mapping policies discussed in Section 2. Table 2 lists the primary configuration parameters of the 8 different 4 cluster machine models that we focus on in this paper.

Machine Model	Cluster Composition #	FUs	Regs	Cluster Latency	Memory	Mapping Policy	Memory Latency min	max
SC-CL1	1	3	64	NA	Centralized	-	6	6
SC-ALLIDEAL	1	12	256	NA	Ideal	-	1	1
MC-IDEALMEM	4	3	64	1	Ideal	-	1	1
MC-CL1	4	3	64	1	Centralized	-	6	6
MC-DL1-CI-LOW	4	3	64	1	Distributed	shared-tag	3	7
MC-DL1-LM-LOW	4	3	64	1	Distributed	indep.-tag	3	7
MC-DL1-CI-HIGH	4	3	64	1	Distributed	shared-tag	3	15
MC-DL1-LM-HIGH	4	3	64	1	Distributed	indep.-tag	3	15

Table 2: Machine Configuration Details

Baseline machine models: We have two baseline machine models that define the minimum and maximum performance that we expect from a clustered VLIW architecture. The SC-CL1 baseline machine model is a single cluster with a centralized L1 cache. The cache takes 6 cycles and the register file takes 2 cycles to access. This is a realistic model and defines the minimum performance we should get over all organizations.

The SC-ALLIDEAL machine model we consider is a single cluster machine with lavish resources but ideal memory and register access latencies of 1 cycle each. This is in a sense the machine with ideal performance. We give a single cluster as much resources that all the 4 clusters have put together and further make the access times to the register file and memory ideal. So in essence it is a 4 cluster machine with zero communication latency for inter-cluster and cluster-memory accesses and the best structure access times. So we have paid nothing for clustering the machine and hence it should achieve the best performance over all the configurations. It would be very difficult to build this machine at the expected clock rate because of the sheer number of ports and the size of the register file.

Realistic Inter-cluster communication, Centralized L1: The MC-IDEALMEM machine is our first multiple cluster machine. The MC-IDEALMEM machine model is one step away from the ideal clustered machine. We pay only for the inter-cluster communication whereas the communication to any memory bank is instantaneous and the access latency of a memory bank is 1 cycle. After detailed analysis of the layout we concluded that all inter-cluster communication takes 1 cycle and so the total time to access a register file is 3 cycles.

The MC-CL1 machine model also has a centralized L1 but accounts for the realistic latencies involved in accessing the monolithic L1 cache and the access and communication latencies associated with remote register writes.

Realistic 4 cluster distributed L1 machine models: In the MC-DL1 organizations each cluster has a local L1 cache bank. Since the local bank is closer to the cluster the access time is only the single bank access time and not the worst case bank access time. The time to access a local bank is only 3 cycles as opposed to 6 cycles. Each local L1 bank has 2 ports and they can be considered to be a local port and a remote port. There are two possible memory mapping strategies that we can implement here - cache line interleaved mapping, and locally managed caches. We call these two organizations MC-DL1-CI and MC-DL1-LM.

We divide each of these into two sub-organizations with low latency cluster-memory interconnect (MC-DL1-CI-LOW) and high latency cluster-memory interconnect (MC-DL1-CI-HIGH). All cluster-memory accesses in both memory mapping strategies with low latency interconnect take between 3 and 7 cycles. After detailed analysis of the layout we concluded that all inter-cluster communication takes 1 cycle and so the total time to access a register file is 3 cycles.

The MC-DL1-LM-HIGH and MC-DL1-LM-LOW machine models we have are identical in all respects to the previous two organizations except for the memory mapping policy. In these organizations each cluster has a locally managed L1 cache bank as opposed to a cache line spanning all the banks. The inter-cluster and cluster-memory latency ranges are identical to the previous two organizations.

6 Results

In order to evaluate the different memory configurations and scheduling algorithms, we simulated the benchmarks shown in Table 3. Our compilation toolchain is a large system consisting of many different base software packages. Because of the complexity of integration, we are just beginning to compile large applications. As a result, our benchmark suite consists of several of scientific kernels, as well as various SpecFP benchmarks. We expect to be compiling the entire SpecFP benchmarks suite in the near future.

Benchmark	Description
micro-rbsor	SOR microbenchmark (C)
micro-rbsorf	SOR microbenchmark (FORTRAN)
micro-vvmul	Vector-vector multiply microbenchmark
mgrid	SPECfp2000 multigrid solver
nasa7	SPECfp92 kernel set
swim	SPECfp2000 shallow water modeling
tomcatv	SPECfp95 mesh generation
wupwise	SPECfp2000 quantum chromodynamics

Table 3: Benchmarks.

In order to keep simulation times reasonable, we ran each benchmark for approximately one billion cycles, which corresponds to roughly 10 seconds of execution time on a 1GHz PC running linux. Of course the actual simulation time depended on the simulated machine and memory model. We also gathered numbers using both instruction schedulers: Desoli's scheduler, and our own alignment-aware scheduler.

Table 4 shows the instruction distribution across all clusters for machine 2bl using the two schedulers. The distributions shown for this machine model are consistent with the other four clustered models. Recall that the alignment-aware scheduler makes no concerted effort towards load balancing. Memory operations are simply assigned to clusters based on their bank alignment. Def-use and use-def information is then used to place the remaining instructions. Using this simple approach, we are able to achieve a noticeable improvement over a state-of-the-art instruction partitioner.

Figure 4 shows the percentage of load instructions that access the local bank. As expected, the alignment-aware scheduler does very well. In most cases, the vast majority of memory operations reference the local bank.

Finally, Figure 5 shows the IPC for all machine models and both schedulers. The first interesting result

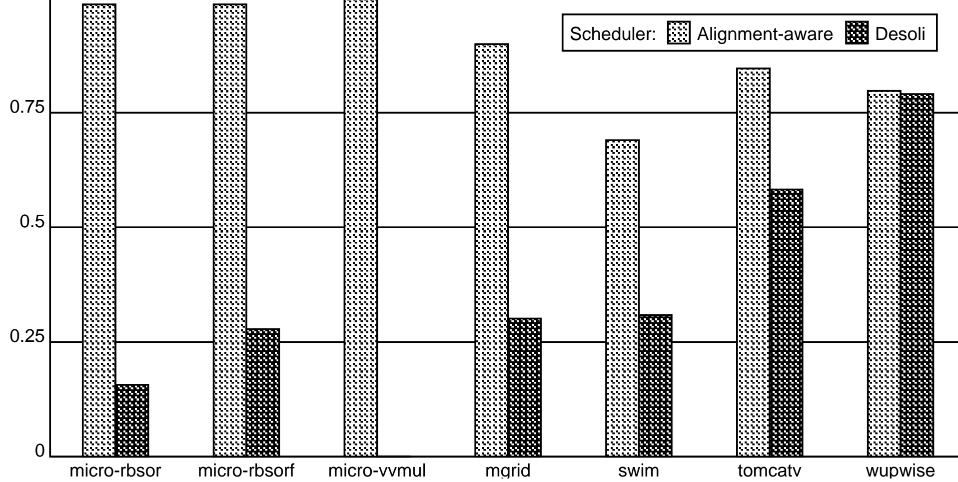

Figure 4: Percentage of local load operations.

is that memory configuration does not have as large an impact as expected. In fact, the difference in performance between a realistic monolithic cache and an ideal memory is rather small. Two phenomenon contribute to this effect. First, the list scheduler used by both partitioning algorithms is reasonably effective at overlapping long latency memory operations with useful instructions. Secondly, we expect only about 25% of memory operations to be loads, limiting the achievable speedup.

When the latency of remote memory operations is increased, the performance of the alignment aware scheduler is affected very little due to our ability to limit global memory traffic. As expected, the performance of the traditional scheduler is degraded drastically as global communication becomes more expensive. Alignment techniques tolerate longer remote memory latencies and reduce global contention so that a cheaper communication network can be used without degrading performance.

It is interesting to note the high performance of the interleaved cache with independent tags. This

micro-rbsor	27.27	23.78	24.24	24.71	49.94	23.33	23.79	2.94
micro-rbsorf	26.71	23.43	24.71	25.14	48.64	21.87	23.52	5.98
micro-vvmul	22.76	25.75	25.75	25.75	79.27	20.73	0.00	0.00
mgrid	53.94	15.78	14.85	15.43	42.20	25.62	19.92	12.26
swim	33.18	20.46	20.46	25.90	37.18	23.57	22.28	16.97
tomcatv	34.76	20.90	21.56	22.78	37.26	22.55	23.56	16.63
wupwise	49.14	27.33	11.28	12.25	57.11	17.17	12.58	13.15

Table 4: Instruction distribution across all clusters for machine 2bl using both schedulers.

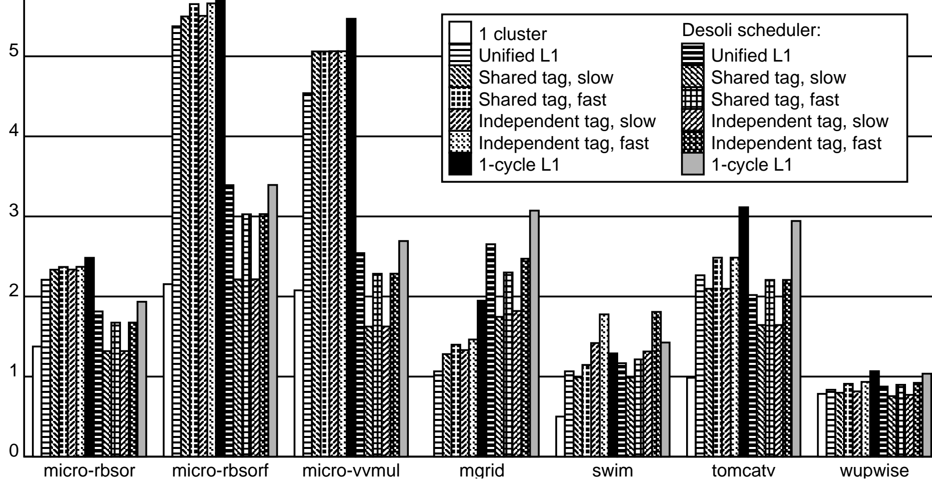

Figure 5: IPC for all machine models and both schedulers.

occurs for benchmarks that iterate through memory with a unit stride. If the compiler is successful at scheduling loops evenly across the clusters, it is likely that the four clusters will access four consecutive locations simultaneously. When the tag array is shared, only a single cache line of data will be fetched for replacement. With our independent tag scheme, four separate lines will be fetched, resulting in greater reuse from spatial locality. Essentially this serves as a type of hardware prefetch for accesses of unit stride.

For this paper, we have limited our study to VLIW architectures with four clusters. With larger numbers of clusters, the trends outlined in this paper would amplify. While previous systems have demonstrated the benefits of up to 28-wide issue [6], the viability of such machines, even with alignment analysis may be limited to particular applications.

7 Related Work

While ample research has been done on scheduling for VLIWs with partitioned register files, scheduling for partitioned memories has largely been neglected. To the best of our knowledge, Fisher [10] and Ellis [9] were the first to discuss the importance of alignment information. They used loop unrolling as a method for increasing the number of aligned memory references. Their work was done in the context of the Bulldog compiler that targeted a clustered VLIW. In their architecture, main memory was distributed across a set of banks. Alignment was important because local memory accesses had lower latency than remote accesses.

In addition, each bank could be accessed in parallel, provided that every cluster accessed a local bank. The use of a pre-loop was also proposed to ensure aligned references in cases where an array base is unknown. However, this transformation was done by hand and apparently only used in simple cases. This research did not propose a mechanism for choosing the exit condition when a loop body contained several references, each with different alignments.

In order to determine which bank was accessed by a particular memory reference, a constraint-based system called *Memory Bank Disambiguation* was used. In order to be successful, this system required the programmer to provide hints about the alignment of certain variables. Comparatively, our alignment detection algorithm requires no programmer intervention in order to detect aligned references.

Barua et al. proposed a more complicated form of loop unrolling [5] to aid in compilation for the Raw machine [20]. The Raw architecture is composed of a mesh of identical tiles, each with a local memory bank. In this design, data access time is a function of the distance to the bank containing the data. Unrolling was used to create memory references that were guaranteed to access a single bank. Precise equations were presented to determine the unroll factors of arbitrary loop nests.

8 Conclusion

The increasing importance of wire delay in high performance, high clock rate systems requires partitioning to minimize global communication. In this paper, we have focused on the challenges of implementing a clustered VLIW architecture, focusing primarily on a partitioned memory system. We proposed interleaving the addresses across the cache banks to allow for cooperative optimization between the architecture and the compiler. Partitioning the banks reduces best-case load latency while knowledge of interleaving allows static analysis reduce the number of remote memory accesses. Our *Alignment Analysis* approach simultaneously optimizes for load balance and minimal memory latency across the clusters, and enhances the power of modern state-of-the-art schedulers. We evaluated two tag management schemes, *shared-tag* and *independent-tag*, which provide different spatial locality benefits for clustered systems.

Alignment analysis was able to increase the fraction of local memory accesses from 34% to 88%, thus

reducing average memory latencies, global communication, and contention. When applied to a memory system with small differentials in local versus remote access time, the overall performance benefits are minor. However, when accounting for estimated contention and longer latencies that will be seen in future technologies, this approach achieves substantial speedups of greater than 33% over a clustered architecture without the benefits of alignment analysis.

This cooperative approach to program optimization will be critical for future performance enhancements. With architectures nearing the limits of pipelining, future performance must come from extraction of greater concurrency. Staying on the historical performance growth curve will likely require at least a factor of 30 increase in concurrency over the next decade. Combining software techniques that expose concurrency with hardware techniques for efficiently exploiting it will help continue to increase performance in future systems.

References

[1] http://www.eecs.harvard.edu/hube.

[2] V. Agarwal, M. Hrishikesh, S. W. Keckler, and D. Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 248–259, June 2000.

[3] V. Agarwal, S. W. Keckler, and D. Burger. Scaling of Microarchitectural Structures in Future Process Technologies. Technical Report TR2000-02, Department of Computer Sciences, The University of Texas at Aust in, Austin, TX, February 2000.

[4] J. R. Allen and K. Kennedy. Automatic Loop Interchange. In *Procedings of the SIGPLAN Symposium on Compiler Construction*, pages 233–246, Montreal, Quebec, June 1984.

[5] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Memory Bank Disambiguation using Modulo Unrolling for Raw Machines. In *Proceedings of the Fifth International Conference on High Performance Computing*, Chennai, India, Dec 1998.

[6] R. P. Colwell, W. E. Hall, C. S. Joshi, D. B. Papworth, P. K. Rodman, and J. E. Tornes. Architecture and implementation of a VLIW supercomputer. In *Proceedings on Supercomputing '90*, pages 910–919, New York, NY USA, November 1990.

[7] W. J. Dally and J. W. Poulton. *Digital Systems Engineering*. Cambridge University Press, Cambridge, United Kingdom, 1998.

[8] G. Desoli. Instruction Assignment for Clustered VLIW DSP Compilers: A New Approach. Technical Report HPL-98-13, Hewlett Packard Laboratories, Jan.

[9] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, Cambridge, Massachussetts, 1985.

[10] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel Processing: A Smart Compiler and a Dumb Machine. In *ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 37–47, June 1984.

[11] S. Gupta, S. W. Keckler, and D. Burger. Technology Independent Area and Delay Estimates for Microprocessors Building Blocks. Technical Report TR2000-01, Department of Computer Sciences, The University of Texas at Aust in, Austin, TX, February 2000.

[12] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, 1, February 2001.

[13] M. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. W. Keckler, and P. Shivakumar. The Optimal Logic Depth Per Pipeline Stage is 6 to 8 FO4 Inverter Delays. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.

[14] S. W. Keckler and W. J. Dally. Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism. In *Proceedings of the 19th International Symposium on Compute r Architecture*, pages 202–213, May 1992.

[15] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[16] S. Larsen, E. Witchel, and S. Amarasinghe. Techniques for Increasing and Detecting Memory Alignment. Technical Memo MIT-LCS-TM-621, MIT Laboratory for Computer Science, Nov. 2001.

[17] D. Maze. Compilation Infrastructure for VLIW Machines. Master's thesis, Massachusetts Institute of Technology, September 2001.

[18] J.-M. Parcerisa, J. Sahuquillo, A. González, and J. Duato. Efficient Interconnects for Clustered Microarchitectures. Technical Report UPC-CEPBA-2001-15, European Center for Parallelism of Barcelona, November 2001.

[19] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An Integrated Cache Timing, Power and Area Model. Technical report, Compaq Computer Corporation, August 2001.

[20] M. Taylor, J. Kim, J. Miller, F. Ghodrat, B. Greenwald, P. Johnson, W. Lee, A. Ma, N. Shnidman, V. Strumpen, D. Wentzlaff, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw Processor - A Scalable 32-bit Fabric for Embedded and General Purpose Computing. In *Proceedings of Hot Chips XIII*, Aug 2001.

[21] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12):31–37, Dec. 1994.