Compiler-directed Data Partitioning for Multicluster Processors

Michael L. Chu, Scott A. Mahlke
Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI 48109-2122
{mchu, mahlke}@umich.edu

ABSTRACT

Multicluster architectures overcome the scaling problem of centralized resources by distributing the datapath, register file, and memory subsystem across multiple clusters connected by a communication network. Traditional compiler partitioning algorithms focus solely on distributing operations across the clusters to maximize instruction-level parallelism. The distribution of data objects is generally ignored. In this work, we examine explicit partitioning of data objects and its affects on operation partitioning. The partitioning of data objects must consider several factors: object size, access frequency/pattern, and dependence patterns between operations that manipulate the objects. This work proposes a compiler-directed approach to synergistically partition both data objects and computation across multiple clusters. First, a global view of the application determines the interaction between data memory objects and their associated computation. Next, data objects are partitioned across multiple clusters with knowledge of the associated computation required by the application. Finally, the resulting distribution of the data objects is relayed to a region-level computation partitioner, which carefully places computation operations in a performance-centric manner.

1. INTRODUCTION

A major difficulty with the design of future embedded microprocessor systems is that traditional designs do not scale effectively or efficiently due to two interrelated problems: centralized resources and wire delay. Centralized resources, including register files and instruction/data caches, become the cost, energy and delay bottlenecks in a processor as it is scaled to support more computation bandwidth [6, 7]. The second problem is that as feature sizes decrease, wire delays grow relative to gate delays. This has a serious impact on processor designs, as broadcasting control and data signals each cycle takes more time and energy. Wire delays are further exacerbated when a processor system is scaled, as the distance between function units, register files and caches increases, thereby forcing the signals to travel further.

To support scalable design, clustered and tiled architectures have emerged as a preferred architecture style for exploiting instruction level parallelism. A clustered design breaks down the centralized register file into several smaller register files [8, 4]. Each of the smaller register files is geographically distributed and supplies operands to a subset of the function units. The clustered design methodology was embodied by the original Multiflow Trace 300 and is commonly used today in embedded processors, such as the TI C6x, Lx/ST200, and the Philips TM1300. MultiVLIW expanded this work by focusing on alternative architecture strategies for designing scalable distributed data memory subsystems [19].

A natural extension to the clustered architecture is the tiled architecture, where each cluster is an entire processor, such as RAW [20]. The interconnect is limited to one or two dimensional nearest-neighbor arrays to reduce wire length. Common to both clustered and tiled architectures is inherent scalability. By distributing resources, designs can be effectively scaled by instantiating new clusters, and interconnecting them into a regular fabric.

One of the most difficult challenges with clustered architectures is compiler code generation technology. It is the compiler's responsibility to partition computation across the processing resources to achieve effective parallel execution. Partitioning algorithms must slice up program computation graphs to spread operations across the clusters, while minimizing the overhead of intercluster communication. Further, computation partitioning is only one part of the compiler's responsibility. The data memory subsystem is often partitioned for the same reasons as the register file was partitioned [10, 11]. Thus, data objects (scalars, arrays, dynamically allocated objects, etc.) must be partitioned across the distributed data memories in each cluster. The objective is to localize data with its associated computation on a cluster, thereby avoiding the serialization effects of frequent intercluster communication caused by long latency and restricted bandwidth interconnect.

Traditional operation partitioning methods are computation-centric and ignore the effects of the data memory [2, 5, 18, 13]. Either a centralized, multi-ported data cache is assumed, or the system contains distributed hardware-coherent caches. In cost or energy constrained systems, such hardware is generally not available. Thus, simpler hardware in the form of scratchpad memories or partitioned caches is often employed. However, distributing data across these simpler memories and then taking advantage of it in the compiler is not a simple task. Terechko et al. [21] studied the effects of partitioning global values in a clustered VLIW processor. They found that remote accesses for global values accounted for approximately half of the cycle count overhead. They evaluated several different schemes of partitioning data, including unified, round-robin, affinity and 2-pass schemes. They concluded that data partitioning must consider the consuming operations of data objects in deciding on an effective memory placement and minimize the remote accesses required.

This work attacks the problem of data partitioning for multicluster processors and proposes an integrated technique for data and computation partitioning. A hierarchical approach is uti-

lized to break the complex problem down into two simpler subproblems that are solved in a phase-ordered manner. First, a global partitioning of the data objects is performed across the entire application. A simplistic view of the computation operations and data communication is employed during this phase to guide the data partitioning. The objective is to balance the memory demands across each cluster. Following this step, a detailed computation-centric partitioning is performed to partition all the operations. Based on the results of the global memory partition, memory operations are locked into place during this phase. However, all other operations must be assigned a cluster and the appropriate intercluster communication inserted. This strategy is effective because the data partitioning is performed at the full application level, and its affects on all computation operations are considered. Further, the data and computation partitioning is cooperative, thus each clustering decision considers its consequences on other related decisions.

2. BACKGROUND & MOTIVATION

This section provides background on multicluster architectures. In addition, we describe distributed data memories within a multicluster processor and an overview of compilation strategies for these architectures.

Multicluster Architectures. Figure 1(a) presents a generic clustered architecture. Each cluster consists of a tightly connected set of register files (RFs) and function units (FUs). FUs may only address those registers within the same cluster. Transfers of values between clusters are accomplished through explicit move operations that travel through an interconnection network. The clustered design shown in this figure assumes an intercluster communication bus that connects the processing elements together with a fixed bandwidth. Though this assumption is not necessary, it is often made because it simplifies compiler algorithms by removing the need to model network topologies with different connectivities. The clusters themselves may be homogeneous, each containing the same types and numbers of RFs and FUs, or heterogeneous, each having a unique mix of resources. The machine in Figure 1(a) is homogeneous and has two FUs, one memory unit, and one RF in each cluster.

The main new compiler task for a multicluster architecture is to obtain a balanced workload that takes advantage of parallelism available within a clustered machine. The notion of balance on a cluster relates to the resources available on that cluster and the operations scheduled on it. For example, given a machine with two heterogeneous clusters such that cluster 1 has twice as many FUs as cluster 2, a balanced workload would tend to have twice as many operations scheduled on cluster 1 as on cluster 2.

Data is transferred from cluster to cluster via explicit intercluster move operations. Intercluster moves have a non-zero latency and thus can lengthen the schedule. However, if the latency of the move can be overlapped with the execution of other operations, then the intercluster moves may not significantly affect performance. A good partitioning of operations minimizes overall schedule length by simultaneously maximizing the number of operations executed in parallel while minimizing the number of moves that negatively affect performance.

There has been a significant amount of prior research in the area of partitioning for multicluster processors. The first clustering algorithm was the Bottom-Up Greedy, or BUG, algorithm in the Bulldog compiler [5]. BUG occurs before instruction scheduling and greedily assigns operations to clusters in order to minimize estimated schedule length. Chu et al. [3] developed the Region-based Hierarchical Operation Partitioning (RHOP) algorithm, which is a pre-scheduling method to partition operations. In order to produce a partition that can result in an efficient schedule, RHOP used schedule estimates and a multilevel graph partitioner to generate cluster assignments. Our second, computation partitioning pass is a modified form of the RHOP partitioner. Aletà et al. use a graph partitioner similar to our computation partitioning phase, but focus on tightly integrating the clustering algorithm with instruction scheduling and register allocation [1]. In addition to these algorithms, many other previous works [2, 13, 18] developed methods for partitioning computation; however, none have added support for partitioning the data objects of a program and accounting for them when they make their computation partitioning decisions.

Data Memory Distribution. While clustered architectures decentralize and partition the datapath into a more scalable form, the data memory can still become a performance bottleneck. There are two main categories of data memory designs for multicluster architectures: *shared cache* and *partitioned caches*. Designing a multicluster architecture with a shared cache that is accessible from every cluster is not easily scalable beyond 1 or 2 clusters. A shared cache must include enough ports for each cluster yet maintain a low access time, which becomes increasingly difficult as the number of clusters grow. Figure 1(a) is an example of a shared cache, as the memory units of both clusters communicate with a single data memory.

The other possible design method would be to use a fully partitioned cache, and have the compiler partition the data across the caches. In such a design, the address space is partitioned across the caches, and data objects have their home in only one of the memories. This is similar to a scratchpad model, where the data objects are known to exist in a specific data memory. This type of memory design requires a sophisticated partitioning of the data in a balanced and efficient manner. For this paper, we focus on the architectural model of a fully partitioned data memory, as shown in Figure 1(b). Thus, a major compiler task is to partition both the data into separate memories as well as the computation across the clusters.

An obvious middle ground would be a coherent partitioned cache, where each processing element has its own cache, but a strict coherence policy is enforced much like a multiprocessor system. While this design meets the goals of creating smaller, dedicated caches, it increases complexity in adding arbitration and coherence mechanisms between caches. The coherent cache model has the benefit of easing some of the difficulties of the compiler task. However, having a coherence protocol and hardware support in a low-cost embedded domain simply adds too much complexity. In addition, the task of partitioning data objects cannot simply be ignored, as a poor partitioning of the data across clusters can result in more coherence traffic.

Recently, there have been many studies in the area of partitioning data memories in the architecture. Gibert et al. [11] use small low latency buffers as localized storage and dynamically fill them in order to improve performance. However, since a miss in their buffers can always fall back to the L1 cache, there exact partitioning of the data objects is not as important. Other recent work by Gibert et al. [9] partition objects to fast-access but high-power and slow-access but low-power caches in order to save power. Critical objects are placed in the fast cache, while non-critical objects are placed in the slower cache.

Hunter [12] studied data objects for characteristics to place them in specialized SRAM arrays. Her partitioning of the data objects focused more on lowering the memory port requirements

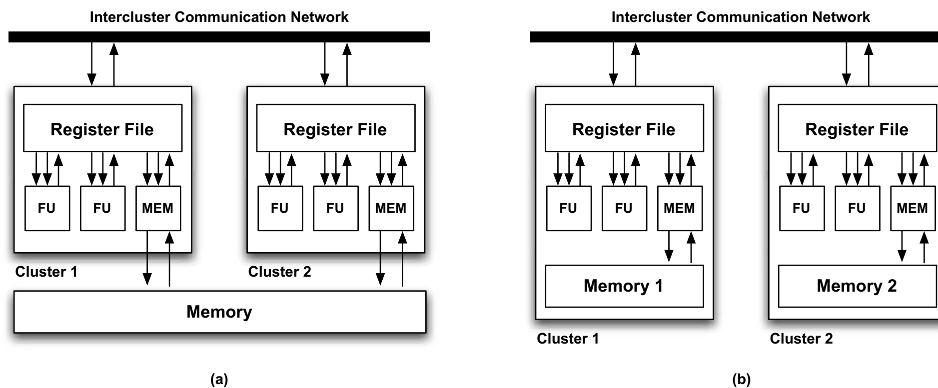


Figure 1: Two multicluster machines. (a) a homogeneous multicluster architecture with a single, unified memory and (b) a homogeneous multicluster architecture with distributed memories in each cluster

and access latencies. The RAW processor is a tiled architecture where certain tiles have the ability to access memory and each tile can only directly communicate with its nearest neighbor. In such schemes, operations in different partitions should be assigned to tiles near each other if they often communicate with one another. The RAW compiler [16] has two phases which first partitions the computation, then places them on tiles near the location of the data they access as well as near the other tiles with which they must communicate data. They partition data by assigning affinities between data objects and instruction streams and group the data objects into sets. While their method is also global, they differ from ours in that they focus more on the affinities of whether or not an instruction stream accesses an object. Our method produces an global graph of the entire program and can consider the communication patterns between data memory accesses and their related computation.

Data Partitioning Issues. The partitioning of data is yet another difficult problem for the compiler to try to optimally solve, as it must consider several factors such as: object size, access frequencies, and dependence patterns between operations which manipulate the objects. In the ideal situation, the objects would be partitioned in such a way to balance memory demands of each cluster, while not hindering performance so that the application performs as if the memory was unified as in Figure 1(a). Thus, the goals of computation and data partitioning are very similar; both hope to generate a partition which has performance of centralized resources on a decentralized processor by reducing communication or hiding communication latency.

Figure 2 shows how partitioning algorithm's assumptions of a shared, unified memory can affect the schedule when data elements are actually placed in distributed caches. Details of the processor configuration and benchmarks are presented later in Section 4.1. For this experiment, each cluster was assumed to have its own memory. For a simple data partition, the actual data is placed in the cache of the cluster which has the most dynamic accesses of that data object. To accomplish this, each static load is marked with the object that it accesses. Objects are placed in clusters by their total dynamic access frequency per cluster. Composite objects, such as arrays or structures, are not allowed to be split across clusters.

The partitioner is allowed to run assuming that the clusters have a shared, unified memory. As a postpass to the clustering algorithm, each object is placed in the cluster with the highest dynamic accesses for the object. Thus, if a memory operation is placed on an incorrect cluster for its data object, the appropriate instructions are put in place to load/store on the remote cluster and then transfer the object across the intercluster communication network. This data placement is not intelligent, as it totally ignores the balance of memory usage across the clusters. In addition, the partitioning algorithm itself is totally incognizant of the data location, so it does not account for the location of the data when making its partitioning decisions. However, it should be fairly performance-centric given the computation partitioning, and highlights the effects of a data incognizant partitioner.

Thus, Figure 2 shows the percentage increase in number of cycles given a 1, 5 or 10 cycle intercluster communication latency. From these results, it is evident that at higher intercluster move latencies the partition of the data has a significant impact on the achievable performance. Partitioning algorithms need to consider where data objects are placed when splitting operations across a distributed architecture. Some benchmarks, such as rawdaudio, had no noticeable difference in performance even at higher intercluster move latencies. This occurred because of other computation-based intercluster moves which were already required that the moves required for data were hidden behind. However, most benchmarks showed little performance loss at 1 cycle move latencies (as the penalty for moving data was almost insignificant) but much more drastic losses at higher latencies. Such large losses in performance suggest that the data memory must be more intelligently partitioned and their locations must be made cognizant to the operation partitioner.

3. GLOBAL DATA PARTITIONING

This section introduces our compiler-directed Global Data Partitioning (GDP) approach for jointly partitioning data objects and computation across a multicluster architecture.

3.1 Overview

In building a general partitioning strategy, we strongly believe that jointly attacking both computation and data partitioning is important in achieving an efficient solution. However, each problem on its own is extremely complex, as partitioning decisions about data affect the decisions on computation and vice versa. Thus, our approach is to break the problem into two simpler sub-problems that are solved in a phase-ordered manner. We believe the best strategy consists of two partitioning phases. An initial memory partitioning is performed to cluster and distribute data objects for an entire application across partitioned memories. This initial partitioning uses a global view of the entire program in order to heuristically model the communi-

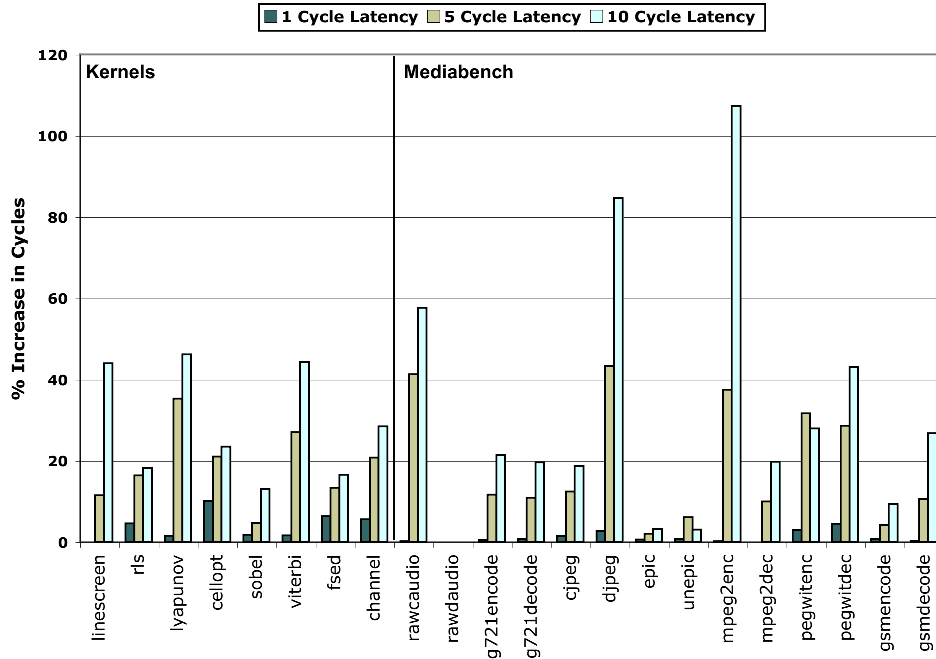


Figure 2: Increase in cycles when data is partitioned across clusters.

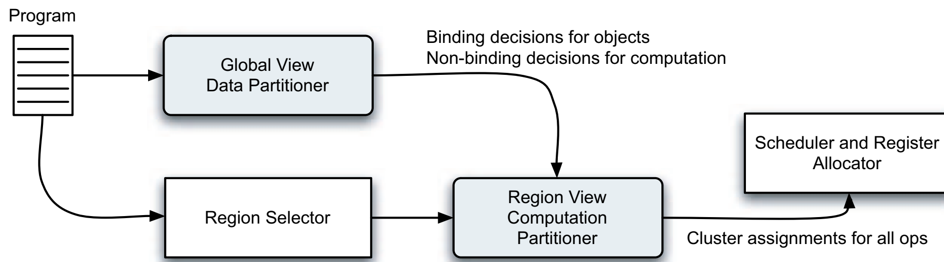


Figure 3: A flow chart of our Global Data Partitioning method, where the new steps are shaded.

cation required for the data memory partition choices. Next, a second, more detailed partition is performed on the computation given a fixed memory placement to finish the distribution process. Figure 3 is a flow chart which shows how these steps fit into a compiler framework. We view data partitioning as a first-order effect; the division of the data across the clusters needs to be decided first to drive the partitioning of the computation. This phase-ordered approach is similar to a common approach used for instruction scheduling and register allocation, wherein prepass scheduling, followed by register allocation, and ending with postpass scheduling is performed. By interleaving the partitioning steps, each has influence on the other in terms of the cost model, but each subproblem is solved in a decoupled manner.

3.2 Prepartitioning Analyses

Before the actual partitioning begins, several compiler analyses are performed in order to determine characteristics of the application. First, the compiler must discover the data memory access locations for each operation. More specifically, each load and store must be analyzed to determine the data objects which can be accessed. For static global data, sophisticated interprocedural analysis (IPA) techniques [17] are used to determine points-to relationships about memory accesses and their related data objects. This analysis assigns a unique identifier (id) to each data object and marks the load and store operations with the data objects that can reach them.

Next, for heap objects, each static `malloc()` call site in the code is given a unique id. Again, the IPA techniques are used to relate static `malloc()` call sites back to load and store objects acting on the heap data. Thus, both static global data and heap data can be assigned unique id's, and their access characteristics can be gathered before the partitioning begins. The compiler builds a data access relationship graph between memory access operations and the data that they can access. Along with this relationship graph, the analyses log the data sizes of each data object either by examining their type sizes for static global variables. In addition, a profile is used in order to determine the amount of data allocated in the heap for each `malloc()` call. The data size information is used to balance the total object size assigned to each cluster during partitioning.

3.3 First Pass: Data Partitioning

The goal of the first pass is to use a coarse-grained view of the code to partition data objects with knowledge of how their distribution across separate memories will affect the future partitioning of computation operations. A high-level view of the computation and communication between operations is used to simplify the problem down for the compiler. Using a very de-

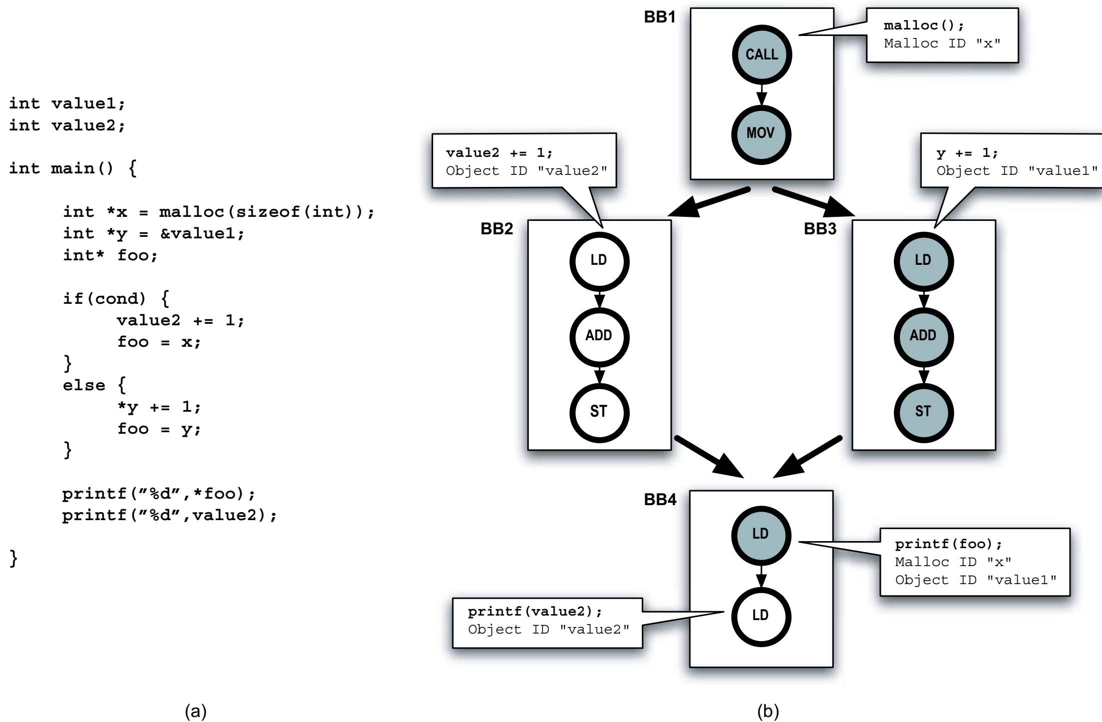


Figure 4: An example of operation merging. (a) the pseudocode for the example (b) the DFG for the pseudocode, where shaded operations are merged together with one another and white operations are merged together.

tailed view of the code schedule, and accurately modeling the data computation partition and the possible effects on the computation and performance, can significantly complicate the algorithm. Thus, a more simplified view of the program behavior is used for the data object partitioning.

First, a program-level data-flow graph (DFG) of the application is created. When creating this graph, nodes are generated from every operation in the code. Memory operations and calls to `malloc()` are annotated in the graph with the ids of their associated objects. This graph is created to generally model the computation patterns that need to be mapped to clusters. The only information recorded about the operations are the data-dependent flow edges. This allows the graph to be partitioned in a way that includes a high-level model of the required computation and intercluster communication traffic.

Figure 4(a) is an example pseudocode with several types of memory accesses. The pointer to "x" refers to dynamically allocated memory in the heap, while the pointer to "y" refers to global data. Depending on a condition, the pointer "foo" is set to "x" or "y" and then accessed at the end of the function. Figure 4(b) illustrates this code is a DFG, only showing the important nodes for this discussion. The interprocedural analysis can determine that the the load and store in BB3 both reference the pointer "y" and that "y" points to the global variable "value1". Similarly, it can find that the first load in BB4 can also access "value1". Profiling of the heap accesses can show that the allocated addresses "x" defined by the `malloc()` in BB1 can also reach the first load of BB4.

After building up the program-level graph, a coarsening process begins, which merges together operations in the graph that would likely prefer to be on the same cluster. This is followed by the actual partitioning of the data objects. The process is described in more detail in the following two sections.

3.3.1 Access Pattern Merges

The access pattern merge phase of coarsening examines the objects accessed by each memory operation and combines operations which access the same memory objects. By merging these memory nodes in the graph, objects themselves become merged. There are two main cases when these data memory objects are merged in this phase. First, when a single memory operation accesses multiple data objects, these objects are merged together. This occurs because the compiler knows that at least one memory operation exists that accesses more than one object, so placing them on separate memories will require data transfers across the communication network. Thus, they are merged together so that they will be placed in the same memory. Second, when multiple memory operations access a single data object, those memory operations will be merged together. Any other objects accessed by these operations will then be merged in as well. These access pattern merges serve to help direct the data partitioner to not unnecessarily break known related objects across separate memories.

For the example in Figure 4(b), since the first load in BB4, could be either "value1" or the allocated region "x", both objects are merged together, and every access to these objects are merged into a single node. The merged nodes are indicated by the shaded operations in the DFG. These include the call to `malloc` in BB1 and the LOAD and STORE of "y" in BB2. Similarly, memory operations in BB2 and BB4 both access the object "value2", so they are merged together, as indicated by the white operations in the figure.

Another possible merging method would be to combine dependent operations with low slack together. This would group together dependent operations into a single unit and potentially combine objects whose computation is highly related. However, in our experimental analysis, we found that merging based on

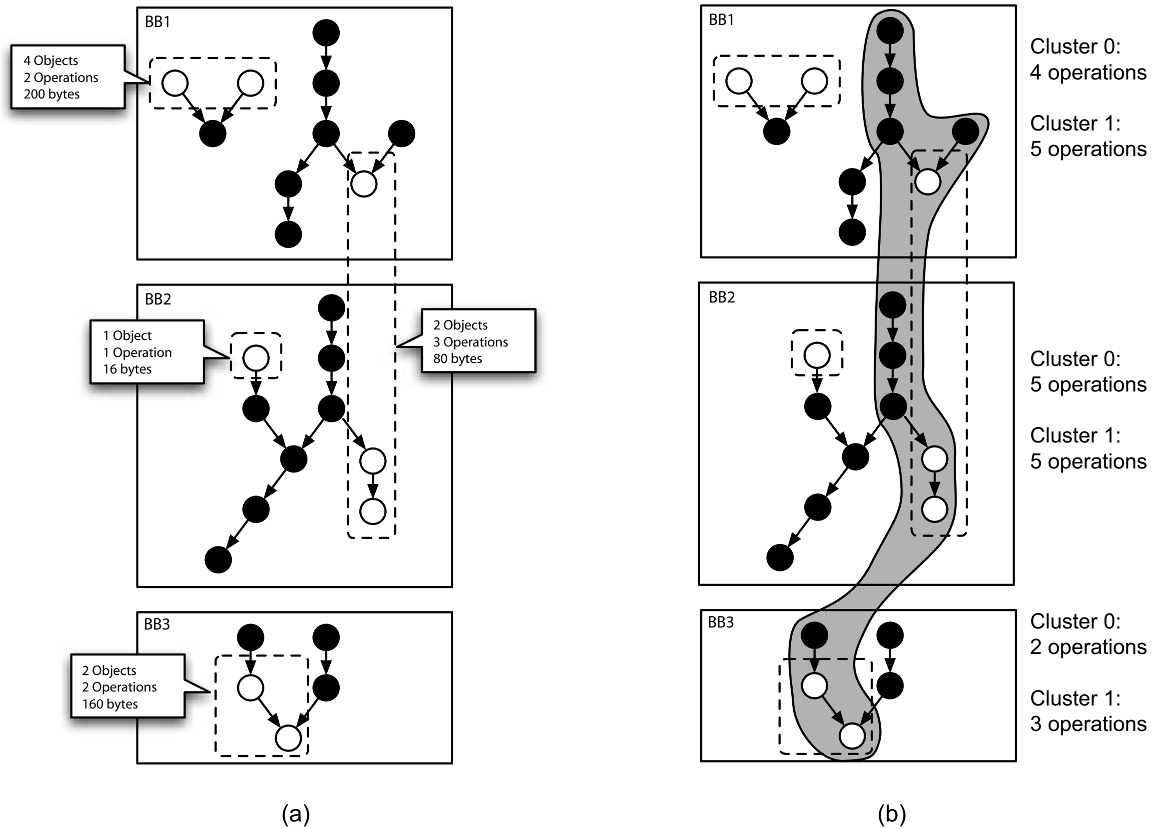


Figure 5: Example of global data partitioning with (a) the original graph with operations coarsened with dotted lines and (b) an example good partitioning, indicated by the shaded region.

computation dependencies can negatively affect the resulting object partitioning. This occurred because fewer groupings of objects allowed for more freedom and flexibility in the partitioning process.

3.3.2 Data Partitioning

After the coarsening process, the compiler is left with a DFG representing the computation of the application, however, some of the nodes have been merged together to form larger nodes. In addition, each node in the graph that accesses data memory is marked with the id of the data object or `malloc()` call site, and the size of the merged data object.

To partition the program graph, we use METIS [14], an efficient graph partitioner which can partition the operations with multiple node weights. METIS tries to divide the nodes into separate partitions by minimizing the number of edges cut while also trying to balance the node weights. The compiler presents METIS with the data-flow graph representing the entire program. Node weights are added to each operation which indicate the size of the data (if any) accessed within that node. This helps the partitioner choose a cluster assignment for the data memory objects that balances the object sizes across clusters. The memory size balance between clusters is parameterized in the case where the memory within one cluster is significantly larger than the other.

Figure 5(a) is an illustrative example of three basic blocks of a DFG which is partitioned by our global data partitioning. White nodes in the graph are memory access operations while black nodes are computation. Dotted lines indicate coarsened operations as explained in Section 3.1. Each grouped memory operation is labeled with the number of objects, the size of the objects, and the number of operations coarsened together. The goal of the partitioner is to balance both the total data memory size as well as minimizing the amount of operation communication cut across clusters, which is indicted in the program-level graph as edge cuts. Figure 5(b) shows an example partitioning produced which could yield such results, where the operations placed in cluster 1 are indicated by the shaded region. In total, the data memory in cluster 0 has 216 bytes of data, while the memory in cluster 1 has 240 bytes of data. In addition, in each of the three blocks, the number of operations on each cluster are balanced fairly well.

3.4 Second Pass: Region-level Computation Partitioning

The second pass of partitioning uses an enhanced Region-based Hierarchical Partitioning (RHOP) [3] in order to distribute computation across clusters given a mapping of data objects to clusters. RHOP is a operation partitioner capable of efficiently generating high-quality operation divisions; however, as with most previous partitioning algorithms for multicluster architectures, it was designed with the model of a single unified memory.

RHOP itself was designed as a performance-centric multilevel graph partitioner for multicluster architectures. The novel aspect with the algorithm was its modeling of the resources and estimates of the schedule length. These were used in order to estimate the schedule length impact of clustering decisions without requiring the need to actually schedule the code, which is a com-

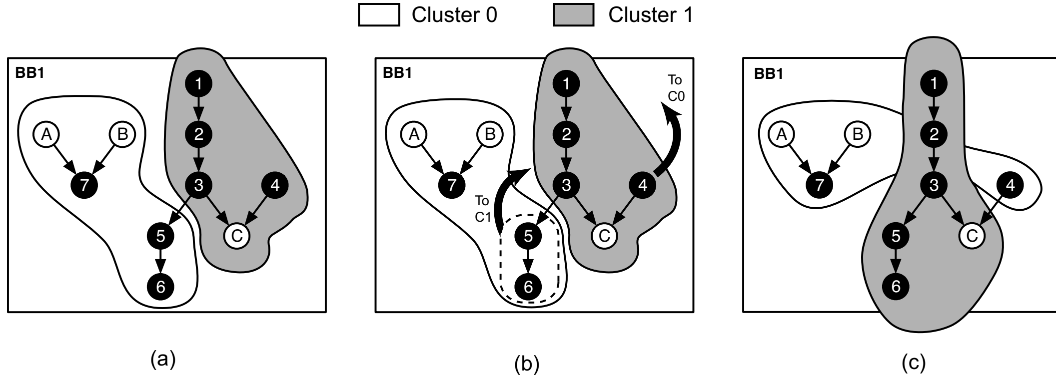


Figure 6: Example of computation partitioning where shaded areas are operations in cluster 1: (a) the cluster assignment designed by the first-pass data partitioning. (b) two performance based improvements made by the computation partitioner and (c) the final partition.

plex and time consuming process. RHOP proceeds by coarsening operations together based on the dependencies between operations. Edges in the graph are given weights based on either low slack between the operations (higher weight), or high slack between the operations (lower weight). A low slack between operations indicates that the edge is more critical, and breaking the edge across clusters will require increasing the schedule length. Similarly, high slack edges have more freedom in inserting intercluster communication. The coarsening process groups together operations in high-weight edge priority. Each stage of the coarsening process groups an operation only once.

After coarsening, the algorithm begins backtracking across the coarsened states, uncoarsening operations. At each stage of the uncoarsening, the schedule length estimates are updated to reflect the current partitioning of the objects. Uncoarsened groups of operations are considered for movement across partitions when they appear favorable in terms of reducing schedule length or resource saturation.

While RHOP has shown to perform well in a single, unified memory case, it requires the data objects be accessible from each cluster. There is no notion of a home location for a data object. Thus, we extended the RHOP method to account for memory object locations in the schedule estimates. When a memory operation is considered for placement in an incorrect cluster, the schedule length estimate would indicate an infeasible partitioning, so that possible clustering choice is ignored. Thus, all memory access operations will always be placed on their assigned clusters, and the schedule length estimators can continue to consider moving other operations for the benefit of balancing computation.

Figure 6 is an example of how the more detailed computation partitioning can improve on the partition produced by the high-level data partitioner. Focusing on only the first block of Figure 5, we see the partition of operations breaking the edge between operations 3 and 5, as shown in Figure 6(a). At a high level, this partitioning seems fairly good, as only one single edge is broken, requiring a single intercluster move and the number of operations per cluster is balanced. During the second pass, memory nodes A and B are, in effect, locked down to cluster 0, and node C is forced to be in cluster 1. However, all other computation operations are free to move about the clusters. Given this ability, the second RHOP pass can note that keeping the critical path 1-2-3-5-6 on a single cluster can be beneficial for the schedule length, and place those operations with memory operation C. Breaking the non-critical edge between 4 and C will not affect the schedule length, so operation 4 is moved to cluster 0 as shown in Figure 6(b). The final partitioning is shown in Figure 6(c). While this creates an imbalance of operations on the shaded cluster, it actually has a better performance because the cluster resources can execute the extra operations in the same number of cycles. Thus, the first pass data-partitioning path works more as a guide, viewing the entire program and dividing up memory usage for the more detailed computation-based second pass.

4. EXPERIMENTAL EVALUATION

4.1 Methodology

We implemented our experimental framework as part of the Trimaran tool set [22], a retargetable compiler framework for VLIW/EPIC processors. We ran our experiments on Mediabench [15] and a set of DSP kernels. Benchmarks were omitted that did not have enough data objects where making a partitioning choice about the memory was important. The machine model used for these experiments is 2-cluster VLIW with 2 integer, 1 float, 1 memory and 1 branch unit per cluster, with latencies similar to the Itanium. Similar to scratchpad memories, partitioned caches that achieve a 100% hit rate are assumed in all experiments. The intercluster network bandwidth allows for 1 move per cycle with latencies of 1, 5 or 10 cycles (5 cycle is default unless otherwise mentioned).

Each benchmark was evaluated for the performance of our Global Data Partitioning (GDP) algorithm compared to three different memory schemes: Profile Max object partitioning, the naïve method shown in Figure 2, and a unified memory model. Table 1 summarizes the differences about these algorithms and they are explained in more detail below.

Profile Max Object Partitioning. In this model, the RHOP partitioner is essentially run twice. The program-level graph of the application is created and coarsened as before, so objects are grouped together the same. The first RHOP pass proceeds to partition the code assuming a single, unified memory, not making any special concessions for the memory objects. Thus, the resulting partition is very performance-centric, as it optimistically assumes each object is accessible from every cluster. After the partitioning is complete, the resulting code distribution is analyzed and the dynamic frequency of each coarsened object being accessed in every cluster is recorded. Then, in order of highest

Algorithm	Object Partitioner	Object Assignment	Computation Partitioner
GDP	Global Data Partitioning		RHOP
Profile Max	RHOP	Greedy (dynamic frequency order)	RHOP
Naïve	None - data object moves inserted post-computation partitioning		RHOP
Unified Memory	N/A - data object moves not required for single, unified memory		RHOP

Table 1: The three different methods tested for object and computation partitioning.

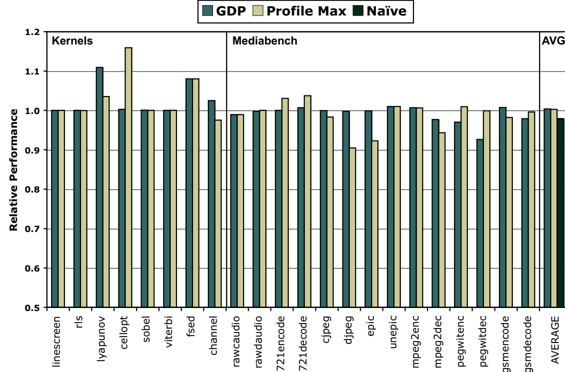


Figure 7: Performance of the GDP and Profile Max methods relative to the single, unified memory for a 1 cycle intercluster move latency.

frequency to lowest, objects are assigned to their preferred cluster (the cluster where the majority of their accesses were placed in the first pass). A memory balance is kept by forcing objects to be placed in other clusters when the preferred memory reaches a certain threshold. Finally, a second pass of RHOP is performed much like the second pass of our global data partitioning algorithm, where RHOP partitions the code cognizant of the object locations. Thus, this is a natural extension to current unified memory clustering algorithms to allow them to greedily partition for multiple memories, and serves as a comparison point for the object partitioning method proposed in this paper.

Naïve object placement. In the Naïve method, as explained in Section 2, no actual object partitioning is performed. As a postpass after computation partitioning, each data object is examined and the frequency of it being accessed on each cluster is recorded. Afterwards, each data object is placed on the cluster where it is accessed most often and required moves for memory accesses are inserted. Note that, in this model, balancing of the memory is not considered.

Unified Memory. In this model, we ignore the case of partitioned memories and simply model a single, multiported memory. Thus, all objects can be uniformly accessed on any cluster in the processor. The unified model represents an upper bound performance because it assumes a constant access latency (2 cycles, the load latency) and no penalty to transfer values across the intercluster communication network. A normal run of RHOP is performed to partition the computation across these clusters. Thus, no preassignment of memory operations is performed on the code. RHOP is simply presented with regions of code one at a time in order to partition the operations. This model can help give an indication of how well the partitioned memories perform in comparison to a unified, shared memory.

4.2 Performance

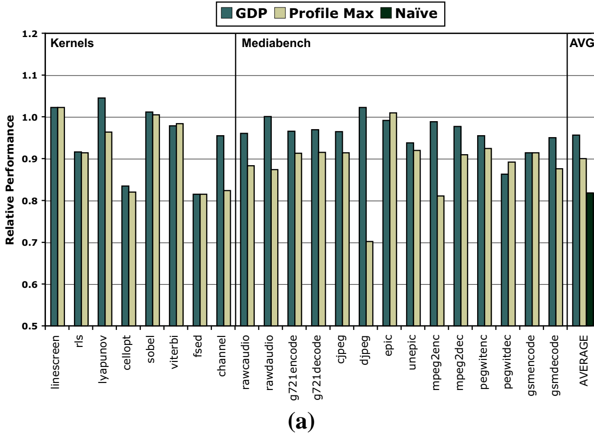
Figures 7, 8(a), and 8(b) show the relative performance of the GDP and Profile Max object partitioning algorithms normalized to the unified memory model with intercluster communication latencies of 1, 5 and 10 cycles, respectively. In addition, the last set of bars in each graph compare the average of these two methods to the Naïve method shown in Figure 2. Higher bars on the graph indicate better performance. Since the graph is relative to the unified memory model, the closer the bar is to 1.0, the closer the partitioned memory is to performing as if it were a single, unified memory.

An interesting fact indicated by these graphs is that in several cases, our partitioned memory is actually performing better than the unified memory case. While the RHOP method, as presented in [3] and used in the unified memory case, is performance-centric, it has one large drawback in comparison to our schemes: its more restrictive view of the program. RHOP already improves upon other previous partitioning algorithms which have a localized, operation-level view of the code when making decisions. However, in our data partitioning method, we take this one step further, from a region-level view to a program-level view for our precoarsening and decision making. Thus, for the second pass, we can give RHOP a better initial partitioning to begin with in order to determine how to improve the computation distribution.

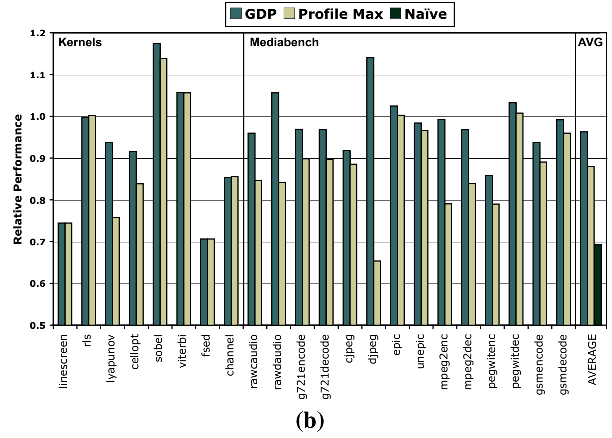
Figure 7 shows the performance given an intercluster move latency of 1 cycle. This graph shows that for most benchmarks, both the GDP and Profile Max methods are able to perform well, and match the performance of a unified memory model. This occurs because with such a low latency penalty for intercluster network traffic, the need to make intelligent object placement decisions becomes less important. Thus, a poor decision on the placement of data will at most only cost only 1 extra cycle to transfer the data to the other cluster. However, such a low intercluster move latency can be unrealistic to build. Thus, we examine higher latency intercluster moves in order to properly gauge the usefulness of data partitioning.

In comparing the 5 and 10-cycle move latencies in Figure 8(a) and (b), our GDP method performs much better than the Profile Max partitioner and achieves near unified memory performance for most benchmarks, and even better in some benchmarks. In the 5-cycle intercluster latency case, our GDP method achieves an average of 95.6% of the performance of the unified cache, while the Profile Max method has an average of 90.0%. For the 10-cycle intercluster communication latency case, the GDP is on average 96.3% of the single memory performance, while the Profile Max scheme is 88.1%. Note that these numbers are slightly skewed because some of the benchmarks achieve more than 100%. However, our partitioning algorithm is able to produce near single, shared memory performance with multiple smaller, partitioned memories. For example, for the mpeg2enc benchmark, at the 1-cycle intercluster move latency, neither the GDP nor the Profile Max methods showed much difference compared to a single, unified memory. Moving to 5 and 10-cycle move latencies, the GDP method was able to maintain 99% performance of the unified memory model while object partitioning of the Profile Max method fell to 81% or 79%, respectively.

Comparing the 5-cycle and 10-cycle latency results shows a



(a)



(b)

Figure 8: Performance of the GDP and Profile Max methods relative to the single, unified memory for (a) 5 cycle (b) 10 cycle intercluster move latencies.

larger gap between the two methods. At higher latencies, intercluster communication has a larger effect on performance. Thus, the quality of the partition in terms of co-locating data and computation is more critical. The Profile Max method is less effective because it cannot account for the inter-region effects of objects and their access patterns. Conversely, the GDP method is make global decisions with a simplified model of the computation and thus can make more intelligent decisions.

In comparison to the data from the Naïve method, both methods did not suffer as much performance loss. This is attributed to the Naïve method being incognizant of the data object locations and simply inserting the necessary intercluster moves as a postpass. Both of these methods are significantly more intelligent as they take the data objects locations into account while performing computation partitioning.

4.3 Exhaustive Search of Partitions

In Figure 9, we present two graphs which represent an exhaustive search of all the possible data object mappings to two clusters for the rawcaudio and rawdaudio benchmarks. An exhaustive search was only possible in benchmarks with a fairly small number of data objects. In both graphs, each point represents the performance of a possible data object partitioning normalized to the worst performing partitioning. The shading of each point indicates the relative data object size balance between the clusters. Darker shaded points are used for more imbalanced partitionings. Thus, the black points are where nearly the entire set of data objects are placed on a single cluster, and white points are where the data objects sizes are well balanced across the clusters. The mappings of both the GDP and the Profile Max method are also marked in each graph.

Figure 9(a) shows the performance of various data mappings for rawcaudio. In this graph, there are many different horizontal bands of object mappings that have relatively the same performance and data object balance. This occurs because there is a small subset of the objects whose cluster placement determine the performance level of the partition. Shifting the other objects between clusters does not greatly affect performance or balance. Both the GDP and Profile Max methods achieved object partitionings which were well-balanced. However, the partitioning chosen by the GDP method had a better performance. While the GDP method was able to choose a good partitioning, the overall performance benefit for rawcaudio was not as impressive, as the best partitioning was still less than 10% improvement of the worst.

Figure 9(b) shows a similar graph for the rawdaudio benchmark. This benchmark has a much more significant performance difference in terms of a good or bad partitioning choice, as the best performing partition choice had almost a 25% performance increase over the worst. Again, this graph shows many horizontal bands. However, at each balance level (indicated by the shading) there is a split in performance at a lower and higher level. This occurs when a small single object can greatly affect performance. Similar to rawcaudio, both partitioning methods were able to find a balanced solution, but the GDP method found a mapping with much better performance. While many points existed with better performance, all had a significantly more imbalanced data sizes, so they were not chosen. Note that the object mappings at better performance, but worse memory balance, can be achieved by allowing for more imbalance of the resulting partition in METIS.

4.4 Increase in Intercluster Traffi c

The quality of a partition can be measured in several ways. One such metric is the number of intercluster moves required during the run of the program. Increasing the number of intercluster moves generally decreases the performance, as more operations must execute, and they all share the communication network bandwidth. However, having more intercluster move operations executing does not necessarily hinder performance. If the intercluster moves can be hidden behind other operations or allow for more parallelization and resource utilization on the clusters, then performance may actually improve. On the whole, however, increased intercluster communication correlates to decreased performance.

Figure 10 shows the increase in dynamic intercluster communication operations for the GDP and Profile Max methods over the single, unified memory processor with an intercluster communication latency of 5 cycles. The baseline processor still has intercluster moves, as it is a multicluster architecture, and requires moves when dependent computation is split across clusters. The GDP and Profile Max methods, however, show intercluster network traffic stemming from both computation-dependent moves and the required transfers of data objects. The most drastic increase in intercluster moves occurs with the fsed kernel. This is correlated with the performance results in Figure 8, as fsed had a large amount of additional moves to insert and had one of the largest decreases in performance.

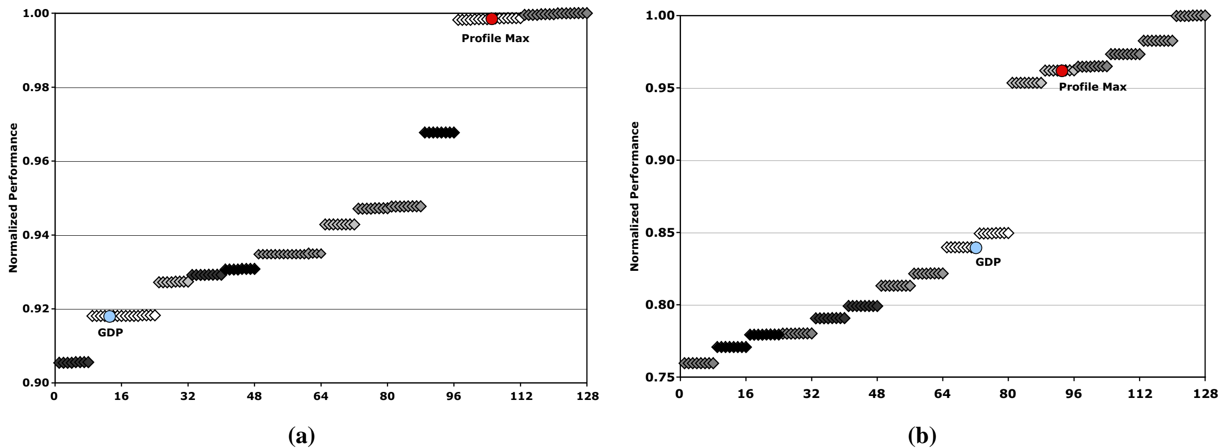


Figure 9: An exhaustive search of all possible data object mappings for the (a) rawcaudio and (b) rawdaudio benchmarks. Points marked with darker shading indicate more imbalanced partitioning in terms of data object sizes per cluster.

For most of the Mediabench benchmarks, the GDP method has far fewer dynamic intercluster move operations executing. In fact, in many cases partitioning the memory has less intercluster traffic than the single memory architecture. This can happen because, again, having a global, program-view prepartition of the data objects can allow the computation partitioner to start with a better initial partition. Of interest is that the benchmarks, such as mpeg2dec and rawdaudio that had a dramatic decrease in dynamic intercluster moves, also have an improved performance in Figure 8.

4.5 Effects on Compile Time

In our experiments, the vast majority of the compiler time was spent in the detailed computation partitioning. The Profile Max partitioner is actually two complete runs of this detailed computation partitioner. The first run is to gather the profile of where objects are placed assuming a shared cache, and the second is to repartition the computation after preplacing objects in their preferred cluster. Since the GDP method only requires one run of this detailed computation partitioner, the compile time is significantly reduced. This is similar to the run time of the Naïve method, which only requires a single run of the computation partitioner.

5. CONCLUSION

In this work, we present a phase-ordered partitioning algorithm which distributes both data objects and computation for multicluster architectures. Partitioning of data objects and computation operations is challenging in that a decision on one can greatly affect the other. Thus, it is important to develop a partitioning method which is cognizant of the side effects of its partitioning decisions. Traditional multicluster partitioning algorithms avoid this problem by assuming a single unified memory for all the clusters, thus simplifying the problem. Our algorithm is a two-phased approach that first partitions the data objects by examining their access patterns at a coarse-grained, program level. By having a viewpoint of the entire program, the data partitioner can make decisions with knowledge of the overall data usage patterns in the program. The second phase, region-based computation partitioning is performed which is cognizant of the preplacement of data objects, and is focused on improving the partition of the computation operations. Overall, our Global Data Partitioning algorithm was able to divide objects across multiple memories yet still achieve, on average, 96.3% of the performance of a single, unified memory memory model.

This current work dealt with partitioning data for a scratch-pad-like, fully partitioned memory model. Our future work will focus on partitioning data objects for cache systems. In order to handle a cache memory system, the partitioning algorithm must be extended to deal not only with communication patterns between data and computation operations, but also with the data usage patterns over time, as objects can be moved into and out of the caches.

6. ACKNOWLEDGMENTS

We thank Hillery Hunter for her advice and feedback on our work. Additional thanks go to the anonymous referees who provided excellent feedback. This research was supported by the National Science Foundation grant CCF-0347411 and equipment donated by Hewlett-Packard and Intel Corporation.

7. REFERENCES

[1] A. Aletà, J. Codina, J. Sánchez, and A. González. Graph-partitioning based instruction scheduling for clustered processors. In *Proc. of the 34th Annual International Symposium on Microarchitecture*, pages 150–159, Dec. 2001.

[2] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *Proc. of the 25th Annual International Symposium on Microarchitecture*, pages 103–114, Dec. 1992.

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

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

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

[6] P. Faraboschi, G. Desoli, and J. Fisher. Clustered instruction-level parallel processors. Technical Report HPL-98-204, Hewlett-Packard Laboratories, Dec. 1998.

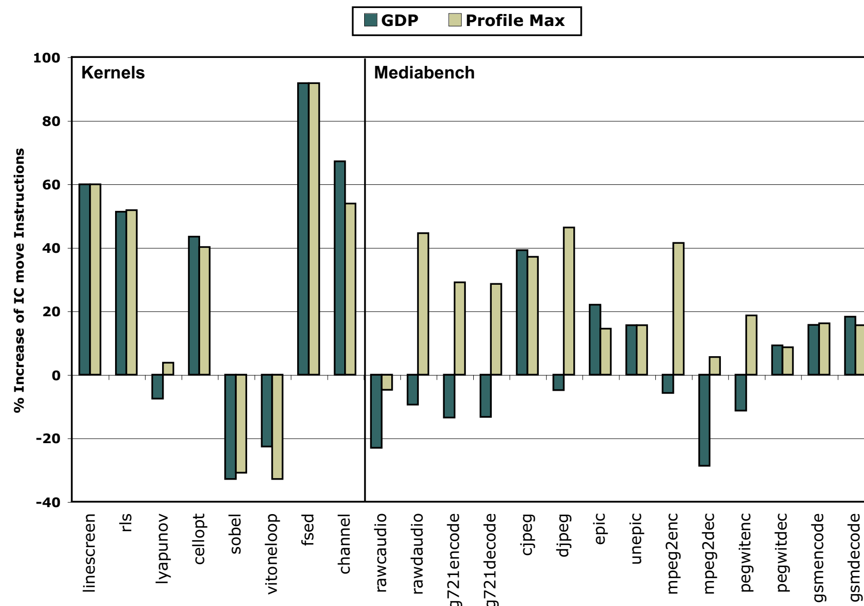


Figure 10: The percentage increase of intercluster move operations using the GDP and Profile max methods over a single, unified memory model with 5-cycle latency intercluster move.

[7] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 149–159, Dec. 1997.

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

[9] E. Gibert, J. Abella, J. Sánchez, X. Vera, and A. González. Variable-based multi-module data caches for clustered vliw processors. In *Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 207–217, Sept. 2005.

[10] E. Gibert, J. Sanchez, and A. Gonzalez. An interleaved cache clustered VLIW processor. In *Proc. of the 2002 International Conference on Supercomputing*, pages 210–219, June 2002.

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

[12] H. Hunter. *Matching On-Chip Data Storage To Telecommunication and Media Application Properties*. PhD thesis, University of Illinois at Urbana-Champaign, 2004.

[13] K. Kailas, K. Ebcioğlu, and A. Agrawala. CARS: A new code generation framework for clustered ILP processors. In *Proc. of the 7th International Symposium on High-Performance Computer Architecture*, pages 133–142, Feb. 2001.

[14] G. Karypis and V. Kumar. *Metis: A Software Package for Paritioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparce Matrices*. University of Minnesota, Sept. 1998.

[15] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, 1997.

[16] W. Lee et al. Space-time scheduling of instruction-level parallelism on a RAW machine. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, Oct. 1998.

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

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

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

[20] M. B. Taylor et al. The Raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[21] A. Terechko et al. Cluster assignment of global values for clustered VLIW processors. In *Proc. of the 2003 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 32–40, 2003.

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