COOPERATIVE DATA AND COMPUTATION PARTITIONING FOR DECENTRALIZED ARCHITECTURES

by

Michael L. Chu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2007

Doctoral Committee:
Associate Professor Scott A. Mahlke, Chair
Professor James S. Freudenberg
Associate Professor Igor L. Markov
Adjunct Professor Steven K. Reinhardt

To my parents.

ACKNOWLEDGEMENTS

The work presented in this thesis could not have been completed without a significant amount of help and support from many people during my graduate career.

First, I would like to thank my adviser, Professor Scott Mahlke, who has helped guide my research directions and develop my abilities to study and investigate new and exciting problems. Scott has been a great mentor and created a great research group which has made my work here enjoyable. I am very lucky to have worked with Scott as a student, teaching assistant and research assistant these past years.

I would also like to thank my dissertation committee, Professor Igor Markov, Professor Steve Reinhardt and Professor Jim Freudenberg, for their time and effort to help improve and refine my thesis. Professor Markov provided me with more feedback and suggestions than I could have asked for. Professor Reinhardt also helped me flesh out the details of my work an push me to investigate new directions. I actually have known Professor Freudenberg for the longest among all my committee members, and I must thank him for working with me as an undergraduate and helping to direct me towards my graduate school career.

The research done in this thesis was done using a compiler infrastructure maintained and support by all the members of the Compiler Creating Custom Processors

(CCCP) group. In particular, I would like to thank Kevin Fan, who helped me develop the slack distribution and weight calculation algorithms for the RHOP partitioner. In addition, Kevin helped me fix countless obscure bugs over the years and also served as my chauffeur more times than I can possibly remember. Rajiv Ravindran also added a significant contribution to this dissertation, helping to devise many experiments for my data partitioning methods, as well as providing great feedback on my papers.

Spending six years in graduate school could very well have been unbearable without many colleagues and friends to make life fun. I would like to thank all the members of the CCCP research group who made my time in the office so enjoyable: Jason Blome, Nathan Clark, Ganesh Dasika, Kevin Fan, Shuguang Feng, Shantanu Gupta, Manjunath Kudlur, Steve Lieberman, Yuan Lin, Mojtaba Mehrara, Robert Mullenix, Pracheeti Nagarkar, Hyunchul Park, Rajiv Ravindran, Misha Smelyanskiy, and Hongtao Zhong. Whether it was throwing around toy HP airplanes, playing Nerf basketball, or xtris competitions, all of you made the office a great atmosphere to work in. I would also like to thank the many other friends who have made my social life away from the office special. I couldn't have finished this without all of you.

Finally, I would like to thank my family for their love and support over the years. My parents have always been there for me and completely supportive of all my decisions. I consider myself very lucky to have so many wonderful people always behind me.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviations

BUG	Bottom-Up Greedy
DFG	Data-Flow Graph
EPIC	Explicitly Parallel Instruction Computing
FU	Functional Unit
GDP	Global Data Partitioning
ILP	Instruction-Level Parallelism
IPA	Interprocedural Analysis
PE	Processing Element
RHOP	Region-based Hierarchial Operation Partitioner
SIMD	Single-Instruction Multiple-Data
TLP	Thread-Level Parallelism
VLIW	Very Long Instruction Word

ABSTRACT

COOPERATIVE DATA AND COMPUTATION PARTITIONING

FOR DECENTRALIZED ARCHITECTURES

by

Michael L. Chu

Chair: Scott A. Mahlke

Scalability of future wide-issue processor designs is severely hampered by the use of centralized resources such as register files, memories and interconnect networks. While the use of centralized resources eases both hardware design and compiler code generation efforts, they can become performance bottlenecks as access latencies increase with larger designs. The natural solution to this problem is to adapt the architecture to use smaller, decentralized resources. Decentralized architectures use smaller, faster components and exploit distributed instruction-level parallelism across the resources. A multicluster architecture is an example of such a decentralized processor, where subsets of smaller register files, functional units, and memories are grouped together in a tightly coupled unit, forming a cluster. These clusters can then be replicated

and connected together to form a scalable, high-performance architecture.

The main difficulty with decentralized architectures resides in compiler code generation. In a centralized Very Long Instruction Word (VLIW) processor, the compiler must statically schedule each operation to both a functional unit and a time slot for execution. In contrast, for a decentralized multicluster VLIW, the compiler must consider the additional effects of cluster assignment, recognizing that communication between clusters will result in a delay penalty. In addition, if the multicluster processor also has partitioned data memories, the compiler has the additional task of assigning data objects to their respective memories. Each decision, of cluster, functional unit, memory, and time slot, are highly interrelated and can have dramatic effects on the best choice for every other decision.

This dissertation addresses the issues of extracting and exploiting inherent parallelism across decentralized resources through compiler analysis and code generation techniques. First, a static analysis technique to partition data objects is presented, which maps data objects to decentralized scratchpad memories. Second, an alternative profile-guided technique for memory partitioning is presented which can effectively map data access operations onto distributed caches. Finally, a detailed, resource-aware partitioning algorithm is presented which can effectively split computation operations of an application across a set of processing elements. These partitioners work in tandem to create a high-performance partition assignment of both memory and computation operations for decentralized multicluster or multicore processors.

CHAPTER 1

Introduction

A major difficulty with the design of future microprocessor systems is that traditional designs do not scale effectively or efficiently due to centralized resources, wire delay, and power constraints. 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 [18, 19]. The second problem is that as feature sizes decrease, wire delays grow relative to gate delays [35]. 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. Finally, in recent years, increased power dissipation and thermal issues have become new first-order design constraints, forcing processor architects to turn towards simpler, more efficient designs.

To support scalable design, decentralized architectures have emerged as a preferred

architecture style for exploiting instruction level parallelism (ILP). An example of a decentralized architecture is a multicluster design, which breaks down the centralized register file into several smaller register files [13, 22]. Each of the smaller register files is geographically distributed and supplies operands to a subset of the functional units (FUs). Thus, a clustered architecture supports distributed ILP across these clusters in order to achieve higher performance. The clustered design methodology was embodied by the original Multiflow Trace 300 [52] and is commonly used today in embedded processors, such as the TI C6x [66], Lx/ST200 [17], and the Philips TM1300. MultiVLIW [60] expanded this work by focusing on alternative architecture strategies for designing scalable distributed data memory subsystems.

A natural extension to the clustered architecture is the tiled architecture, where each cluster is an entire processor, such as RAW [62]. 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. However, the main drawback involved with decentralized designs is the difficulty in compiling code for them. In order to generate efficient code, the compiler must be cognizant of its interactions with the decentralized resources and the side effects of its choices.

In this dissertation, compiler code generation technology, one of the most difficult challenges with decentralized architectures, is addressed. In traditional decentralized designs with a partitioned register file and shared memory, it is the compiler's responsibility to partition computation operations across the processing resources in order

to achieve effective parallel execution. The compiler must carefully weigh the benefits of distributing the available parallelism by splitting the application's operations across clusters with the additional communication overhead required to transfer data between them. This requires a deep understanding of the decentralized architecture and strong compiler analyses to partition the operations of the program across the clustered resources.

The partitioning of computation operations divides up the application in the presence of a decentralized register file, but the compiler may have additional responsibilities. The data memory subsystem is often partitioned for the same reasons as the register file was partitioned [25, 26]. In the presence of a decentralized static local memories, 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 of the interconnection network.

While this research began as an investigation of multicluster partitioning techniques, much of the work can be applicable in multicore processor environments. Multicore processor designs have become a commonplace decentralized architecture in response to growing concerns over power dissipation. These processors help alleviate the power requirement problem by using multiple simpler cores and integrating them on a single die. A major architectural feature of these processors in contrast to clustered and tiled architectures is the presence of distributed caches across the cores rather than a single shared cache. Distributed caches must be carefully taken

into consideration during code generation; a good distribution of the data accesses is critical to producing a high-performance program. A poor distribution of the memory could lead to a large amount of processing time spent stalled waiting for memory due to coherence traffic or cache misses.

The focus of this dissertation is to explore compiler techniques to effectively take advantage of inherent fine-grain parallelism in application code when presented with decentralized architectures. As architecture designs have continually distributed more centralized resources within the processor, the compiler challenges have increased. Multicluster and multicore processors can decentralize register files, data memories, and interconnection networks, all of which increase the complexity of code generation. Common to all decentralized processors is a need to exploit parallelism in order to achieve high performance. The proposed techniques involve a static analysis method to perform global program-level partitioning of the data objects for a decentralized memory model. This technique works well in the presence of distributed scratchpad and static local memories; however, the static analysis-based partitioning does not translate to data caches well. The coarse, object-based partitioning fails to effectively utilize the cache memory space or the coherence netowrk. Thus, a profile-guided method to distributed data access operations at a fine-grain level is presented, which is beneficial in the presence of distributed data caches. Either technique can be used in conjunction with a proposed hierarchical, resource-aware computation partitioner to divide operations across a set of processing elements, in the presence of a decentralized register file. Thus, the combined partitioners performs a data-cognizant partition of the computation to localize operations near their accessed data. These data memory

and computation partitioning techniques work cooperatively to create a two-pass compiler infrastructure to first distribute data, then partition computation operations across a decentralized architecture.

1.1 Contributions

This dissertation makes the following contributions.

- A compiler technique using static analysis to perform a program-level partitioning of the application data objects across scratchpad memories.
- A profile-based technique for distributing memory access operations across multiple distributed data caches.
- A resource-cognizant graph partitioning algorithm for dividing computation operations across multiple processing elements.
- A cooperative two-phase compiler framework for decentralized processor code generation: global partitioning of the data memory which helps to drive the partitioning of the computation
- An method for extracting fine-grain threads, which can be divided across multiple processing elements and executed in parallel.

1.2 Organization

The remainder of this dissertation is organized as follows. Chapter 2 provides brief overview of multicluster and multicore processor designs and the difficulties involved in their compilation process. Chapter 3 presents a static analysis approach for distributing data and computation across partitioned scratchpad memories. With partitioned scratchpad memories, each processing element has its own local memory associated with it, and the compiler must statically place each data object in one of the memories. A profile-guided method for handling distributed data caches is presented in Chapter 4. This technique uses a profile of the memory accesses to determine affinity relationships between memory operations and intelligently distribute them across the caches. The method for purely partitioning computation across multiple clusters is presented in Chapter 5. This method assumes a shared, data memory that is accessible from each cluster with uniform access latency. The method for combining data and computation partitioning into a single phase-ordered process is explained in Chapter 6. Finally, conclusions and future directions are presented in Chapter 7.

CHAPTER 2

Background & Motivation

2.1 Multicluster Architectures

Superscalar and Very Long Instruction Word (VLIW) processors achieve high performance by exploiting ILP to issue multiple operations each cycle. As the number of operations issued each cycle grows, the demands to supply operands to these operations also increases. In a conventional processor, a centralized register file is responsible for operand supply. Supplying a larger and larger number of operands each cycle from a centralized register file can quickly become a bottleneck in a processor design. The bottleneck results due to the combined effects of: register file cost and access time growing with the square of the number of register ports; a larger number of registers being necessary as issue width increases to maintain more temporary values; register bypass logic growing quadratically with the number of operations issued per cycle; and the distance separating function units (FUs) from the register file increasing with a larger number of FUs [18, 19].

A natural solution to these problems is to remove the centralized register file and create a decentralized architecture with several smaller register files. Each of the smaller register files supplies operands to a subset of the FUs. These smaller register files can be efficiently designed, thereby alleviating the register file bottleneck while maintaining the desired level of ILP. This strategy is generally referred to as a clustered architecture or a multicluster processor [22]. One of the first clustered architectures was the Multiflow Trace [52]. Clustered architectures are becoming increasingly popular in many recent processor designs including the Alpha 21264 [29], TI C6x series [66], and Analog TigerSharc [23] and Lx/ST200 [17]. Each of these processors is a two-cluster design.

Figure 2.1 shows the general progression of decentralized architectures. Figure 2.1(a) presents a generic centralized processor with a shared register file and a shared memory. The register file has a large number of read/write ports, which forces a slower access latency; however, each FU can read from the register file in each cycle, which can increase throughput. Figure 2.1(b) presents the traditional decentralized clustered architecture, with multiple register files. Each cluster consists of a tightly connected set of register files and 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. Thus, transferring a scalar value from one cluster to another requires some additional non-zero latency.

Similar to the register file, the data memory itself can become a large, centralized resource shared among the clusters. Figure 2.1(c) represents a homogeneous

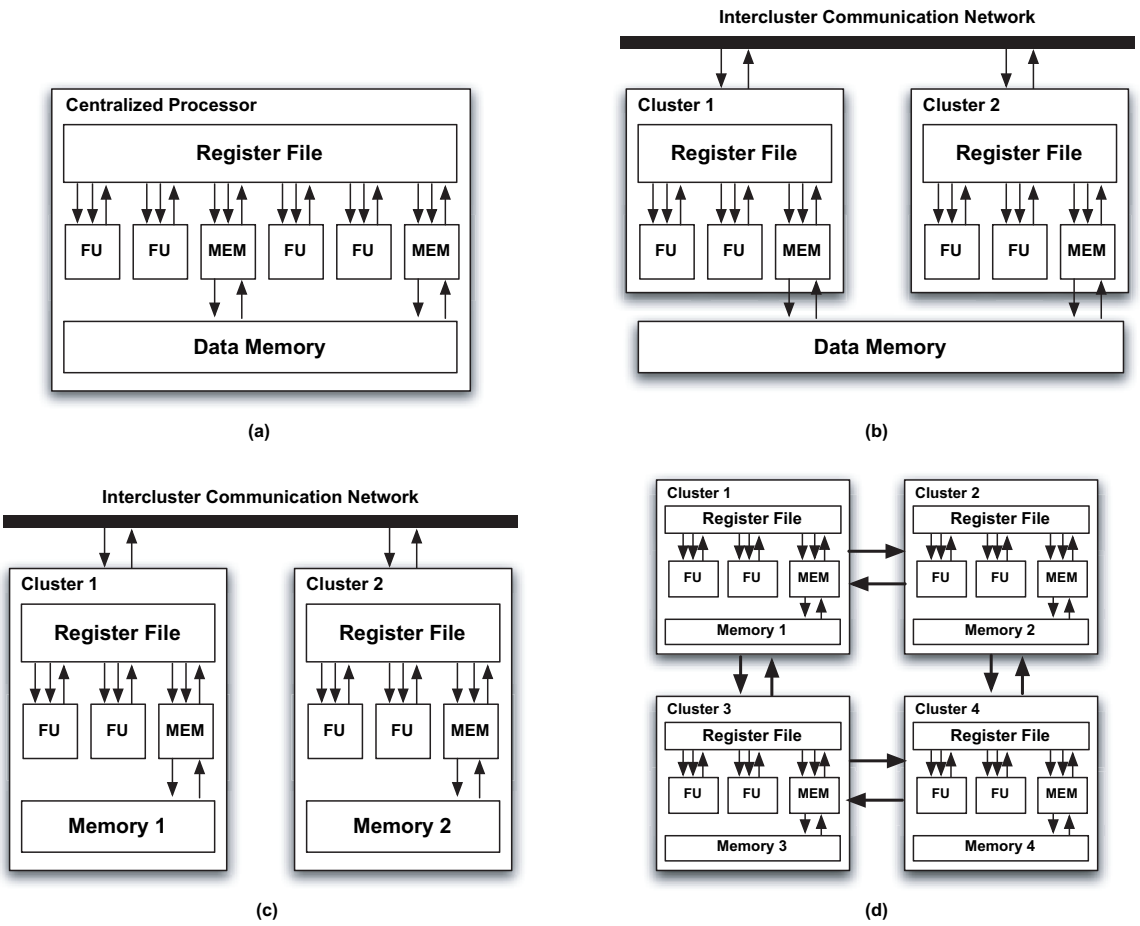

Figure 2.1: The evolution of decentralized processors: (a) a centralized architecture (b) adding a decentralized register file (c) adding a decentralized data memory and (d) a tiled processor with decentralized interconnect.

decentralized architecture with a partitioned data memory. For a large number of clusters, a single, unified data memory as shown in Figure 2.1(b) can suffer the same performance drawbacks of a larger number of ports and slower access time. By decentralizing the data memory, smaller, more efficient memories can be used, but this may again come at the cost of increased compiler complexity. Partitioned data memories can exist as either scratchpad memories, where data objects are statically placed by the compiler, or as caches, where the data objects are dynamically brought into the

memories at runtime. The address space can either by partitioned across the data memories or each memory can have its own address space.

The clustered designs shown in Figures (b) and (c) assume 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. Figure 2.1(d) shows a tiled architecture with a decentralized interconnection network. Data can be transferred between the clusters, but some transfers may take multiple hops to reach its destination.

In the embedded-processor domain, the multicluster architecture has evolved over the years to include more decentralization in their design. While the throughput and capabilities of these processors have continually increased, their actual achieved performance relies on compiler code generation to effectively use the underlying resources.

2.2 Multicore Architectures

In the general-purpose domain, multicore architectures have helped improve processor performance by increasing the number of available resources. A multicore processor consists of multiple cores, or processing elements (PEs). Each PE also has its own data cache, and the caches in each PE are kept coherent through a coherence network. Thus, rather than having a single monolithic processor with a fast clock frequency, multicore architectures simplify the individual cores, but increase their

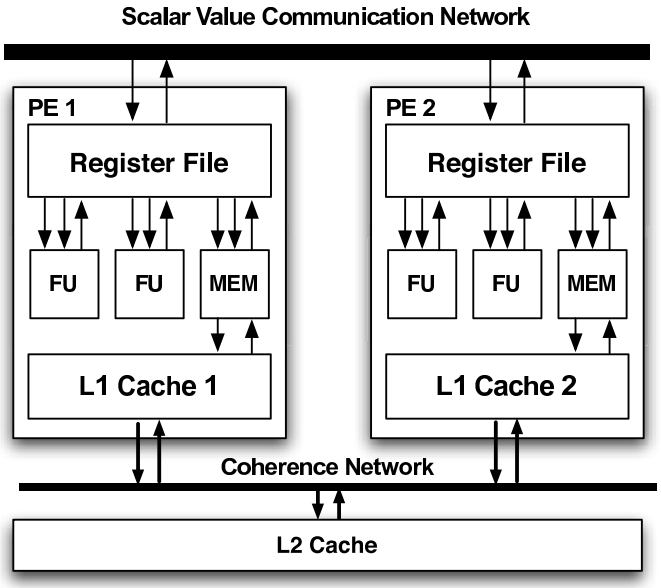

Figure 2.2: A multicore processor with distributed data caches.

number to improve processor throughput. Increased performance in these processors then relies heavily on throughput rather than clock frequency.

A challenging task for multicore processor code generation is exploiting enough parallelism to successfully utilize the available throughput. Recent work in multicore processor interconnects has focused on improving the latency and bandwidth of the communication network [57, 64]. These techniques allow for faster transfer of scalar values between the register files, which can then allow the insertion of communication operations to split producer-consumer relationships across multiple PEs. Thus, fine-grain threads, which are small subgraphs of computation and data access operations can be created to maximize resource utilization.

A generic multicore architecture consisting of two PEs is shown in Figure 2.2. The PEs are connected together by a fast communication network to transfer scalar operand values between PEs, similar to the intercluster communication network in

the multicluster processor. The architecture shown presents another difficulty for code generation because of its distributed data caches. The L1 caches of each PE must be connected to one another with a coherence network to arbitrate the sharing of data lines and maintain correctness. This distributed design allows the processor to be more easily scaled to wider issue designs by simply instantiating new PEs.

Several problems arise with the use of distributed data caches. First, each memory operation now can have a widely varying access latency for its data, depending on where it exists and its current coherence state. The data could reside in its local cache, in a remote cache, or in none of the caches. In each case, the data could also be in shared, modified, exclusive, or other states, each requiring different coherence requests to be sent out to copy from or invalidate other caches, and causing a variable amount of delay because of possible congestion in the coherence network depending on the coherence protocol. Another problem that arises with distributed data memory is that each PE now has a smaller amount of memory associated with it. Compared to the large, shared memory, these smaller L1 caches can have a higher likelihood for cache misses. Thus, the compiler must be careful about what accesses it chooses to place in each cache. Finally, with distributed caches, it is possible to partition the accesses up in a manner in which they cause multiplicative misses in each of the L1 caches, rather than localizing the misses to one of the caches.

The general-purpose domain has seen decentralization of resources as an effective way to balance performance with power constraints. Effective utilization of these resources currently falls on both the programmer to expose parallelism, and the compiler to make use of it. However, many compiler analyses can help in exposing more

parallelism to increase performance and throughput.

2.3 Compiling for Decentralized Architectures

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 [16]. BUG occurs before instruction scheduling and assigns operations to clusters in order to minimize estimated schedule length. BUG traverses the critical path, or longest string of operations, in a region of code and greedily assigns operations to partitions given the schedule impacts of other currently assigned operations. While this can generally provide good solutions, its locally greedy decisions have been shown to fall into local minima with large, complex graphs.

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]. Integrating with scheduling and register allocation can seem logical, since each compiler method is highly dependent on one another; however, each step is itself a very difficult problem. Combining them together can make the problem significantly more difficult to solve. In addition to these algorithms, many other previous works [6, 37, 40, 48, 56] developed methods for partitioning computation. These algorithms all vary in their integration with scheduling and register allocation, and also with the scope of their decision making, from local to regional. However, none have added support for partitioning the data memory of a program

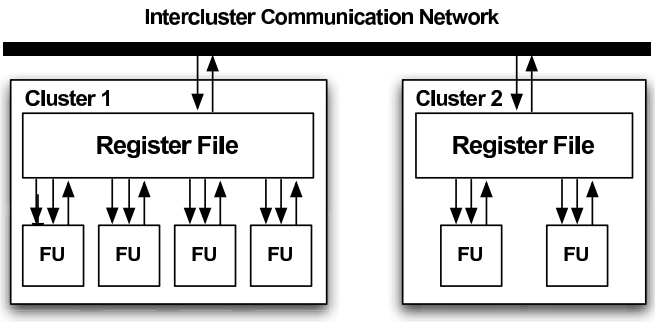

Figure 2.3: A heterogeneous multicluster architecture with twice as many FUs in cluster 1 as cluster 2.

and accounting for its placement when they make their computation partitioning decisions.

The goal of clustering is to obtain a balanced workload that takes advantage of parallelism available within the 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, as shown in Figure 2.3, 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 inter-cluster 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.

In the presence of a partitioned data memory, as shown in Figure 2.1(c), the prob-

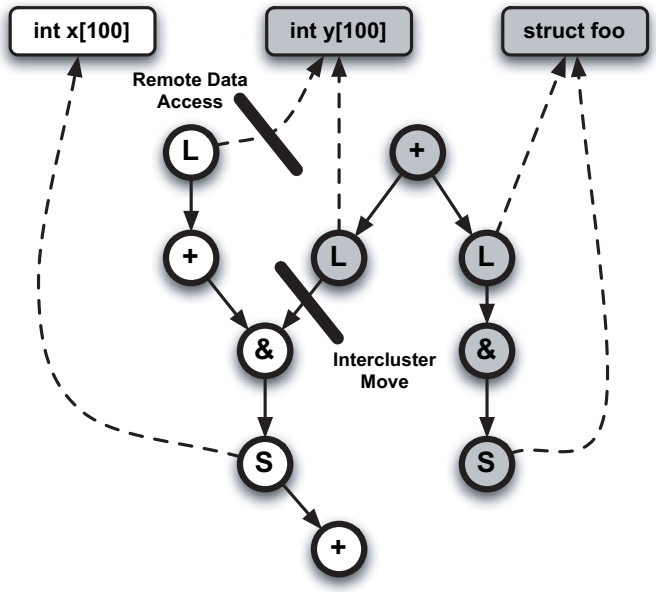

Figure 2.4: An example data flow graph and referenced data objects partitioned among two clusters. White nodes are in cluster 0, shaded nodes are in cluster 1.

lem of clustering data is even more difficult, as the compiler has the additional task of intelligently partitioning the data objects of the program across the memories. Once the objects are partitioned, the compiler must be cognizant of their cluster location when placing load/store accesses to them. A poor decision on object placement can lead to a significant increase in remote loads, in the case of scratchpad memories, or memory stalls, in the case of data caches. Either case could lead to a significant reduction in performance.

Figure 2.4 is an example of a data flow graph partitioned into two clusters. One cluster is indicated by the white nodes, the other by the shaded ones. In addition, the object being accessed by each memory access operation is indicated with a dotted line. For this partition, the two thick dark lines indicate intercluster transfers. In one, the load is accessing an object that is placed in the other clusters data memory. Thus,

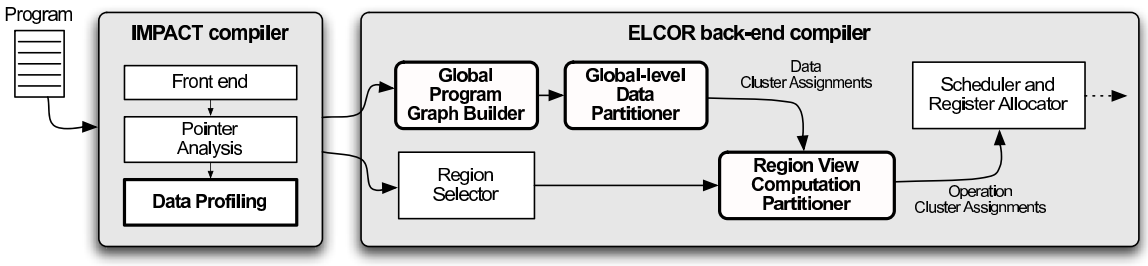

Figure 2.5: A flowchart for our compiler framework for decentralized processors. New compiler phases are shown in bold.

this requires a remote data access, and added latency to execute the load and transfer the data object between clusters. In the second, the AND operation in is consuming a value that was generated in the other cluster. This requires an intercluster move through the interconnection network, and again, added latency. Thus, in order to produce efficient code, the compiler must take all of these transfer latencies into account when producing an operation partitioning for an application.

2.4 Our Compiler Framework

The main contribution of this dissertation proposal is a generalized compiler process to generate efficient code for decentralized processors. Figure 2.5 presents a flowchart for our compiler framework for decentralized processors. The major new compiler phases are shown in bold. In the IMPACT compiler, interprocedural pointer analysis is performed which annotates memory operations in the code with the objects that they reference. This is followed by a profiling phase which collects information about each object, how often it is accessed and characteristics of the memory operation accessing it. Two separate methods for data partitioning are presented, which

provide both a coarse-grain static object partitioning, or a fine-grain profile-guided data access partitioning.

In the elcor back-end, we extended the compiler to build a data-flow graph of the entire program. This allows for a program-level view when making decisions about partitioning data objects. The data partitioning assigns data objects to clusters and is followed by a region-level computation partitioner, which carefully places each operation to a cluster. Finally, the cluster assignments of all operations are passed to the rest of the compiler to perform scheduling and register allocation. All of our extensions are pre-scheduling and phase-ordered, which can free the compiler from the complexity of having to make decisions on schedule time, operation partitioning and object partitioning simultaneously.

CHAPTER 3

Compiler-directed Data Object Partitioning for Multicluster Processors

3.1 Introduction

In the embedded-processor domain, large, centralized resources have continually been the performance bottleneck as architectures have scaled to wider designs with more capabilities. The decentralization started with the register files with the advent of the multicluster architecture, and has now begun to focus on the centralized data memory. A distributed data memory subsystem can benefit the processor design by using smaller, faster memories, but require an additional latency to transfer values between clusters. However, knowledge of an underlying decentralized memory can allow the compiler to be proactive in its assignment of memory access operations to reduce memory transfers.

Traditional multicluster operation partitioning methods are computation-centric and ignore the effects of the data memory [6, 16, 37, 56]. 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 distributed 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. [65] 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 chapter attacks the problem of data partitioning for multicluster processors and proposes an integrated technique using static analysis for data and computation partitioning. A hierarchical approach is utilized to break the complex problem down into two simpler sub-problems 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. This computation partitioning is a slightly modified version of the effective partitioner introduced in Chapter 5. 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.

3.2 Background & Motivation

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

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.

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 chapter, we focus on the architectural model of a fully partitioned data memory. 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. [26] 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. [24] 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. Avissar et al. [3] studied techniques to allocate data

objects across heterogeneous memory units such as scratchpad memories, and internal and external DRAM.

Hunter [36] 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 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 [45] 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 and fit within the capacity constraints of the memory, while not hindering performance so that the application

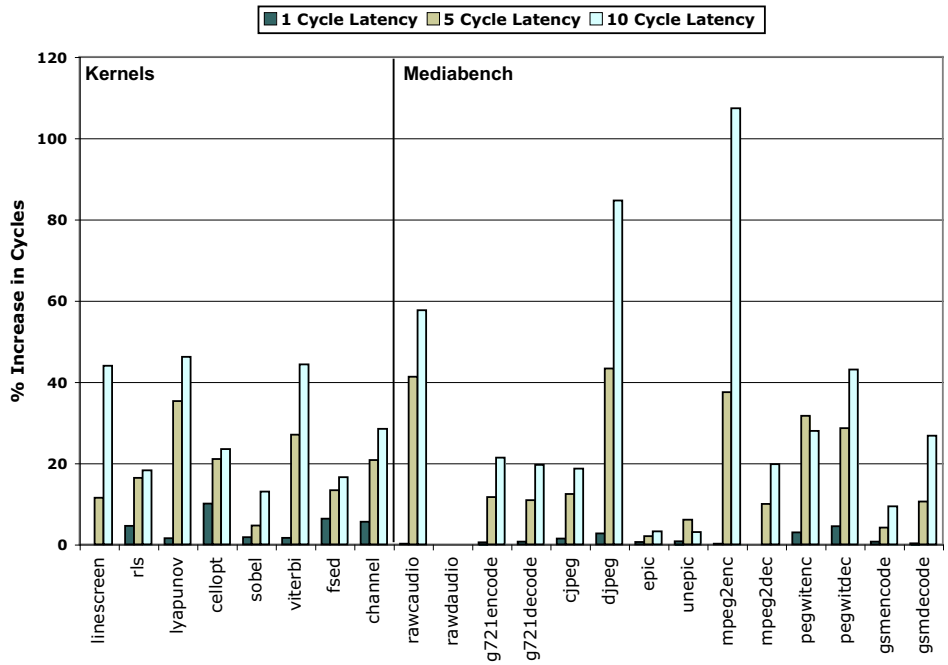

Figure 3.1: Increase in cycles when data is partitioned across clusters.

performs as if the memory was unified. 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 3.1 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 3.1 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

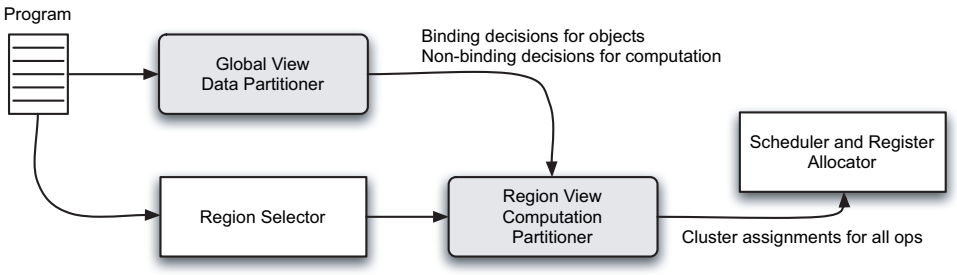

Figure 3.2: A flow chart of our Global Data Partitioning method, where the new steps are shaded.

the data memory must be more intelligently partitioned and their locations must be made cognizant to the operation partitioner.

3.3 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.

3.3.1 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 communication 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 3.2 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.

3.3.2 Problem Definition

The following definitions formulate the graphs and objectives of our data partitioner.

1. **Program Graph:** The graph of the entire application being partitioned, $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, where $\mathbf{V}$ is the set of operations, both memory and non-memory, and $\mathbf{E}$ are the edges, which indicate data exchanged between the operations in a producer/consumer relationship.

2. **Data Objects:** A set of objects, $\mathbf{O} = \{O_1...O_n\}$, which are all the data objects present in the program. Each memory operation in $\mathbf{V}$ accesses one or more

object **O**.

3. **Architecture:** Given **k**, the number of memories, the machine $\mathbf{M} = \{C_1...C_k\}$, where $\mathbf{C_n}$ is a cluster in the processor with a local memory.

4. **Partition Assignment:** A function of the form $\delta : \mathbf{O} \rightarrow \mathbf{M}$, wherein each of the objects of **O** are mapped to a single cluster $\mathbf{C_n}$.

5. **Memory Size:** A value, $\mathbf{S_n}$, which represents the total size of the data objects present in cluster $\mathbf{C_n}$.

6. **Cut:** An edge, e, is cut if, with respect to the partition assignment δ, its source and destination vertices are mapped to more than one memory.

7. **Data Partitioning Problem:** Given a DFG $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, find a partition assignment $\delta : \mathbf{O} \rightarrow \mathbf{M}$ that maps the objects of **O** onto one of the **k** disjoint memories such as the weight of the cut edges is minimized and the memory size, $\mathbf{S_k}$, is balanced.

3.3.3 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 [55] 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.

3.3.4 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 detailed 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

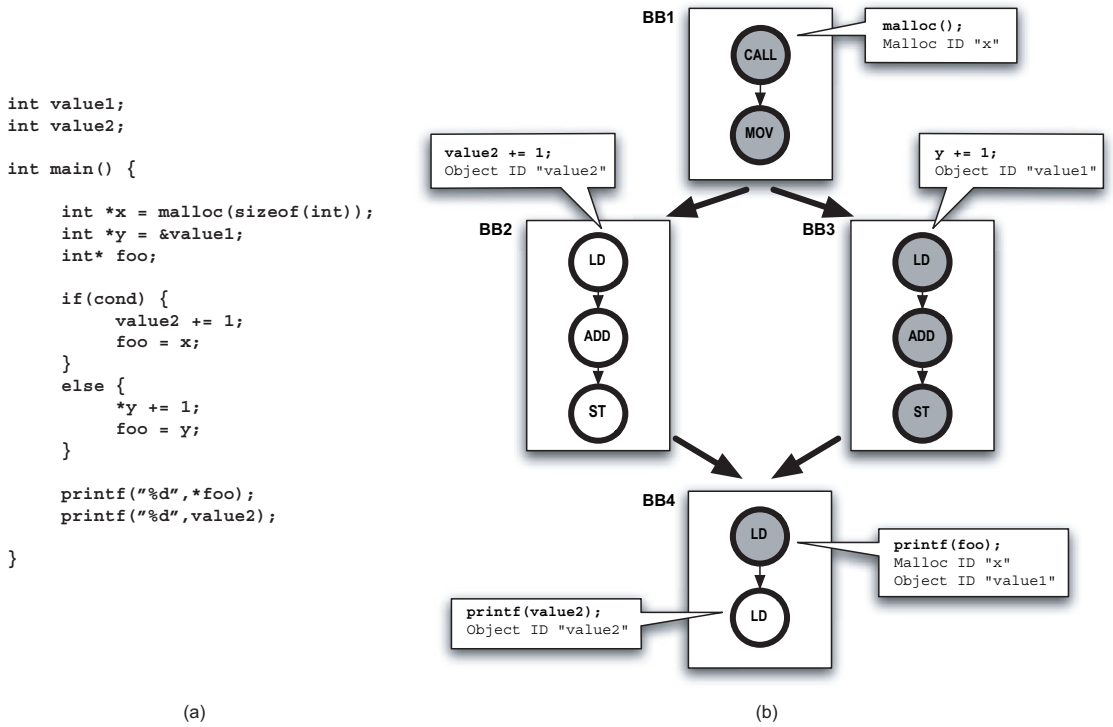

Figure 3.3: An example of operation merging. (a) the pseudocode for the example (b) the DFG for the pseudocode, where shaded operations are merged together with one another and white operations are merged together.

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 3.3(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 3.3(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.

3.3.4.1 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 3.3(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 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.

3.3.4.2 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 [38], 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 3.4(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 3.3.4.1. 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

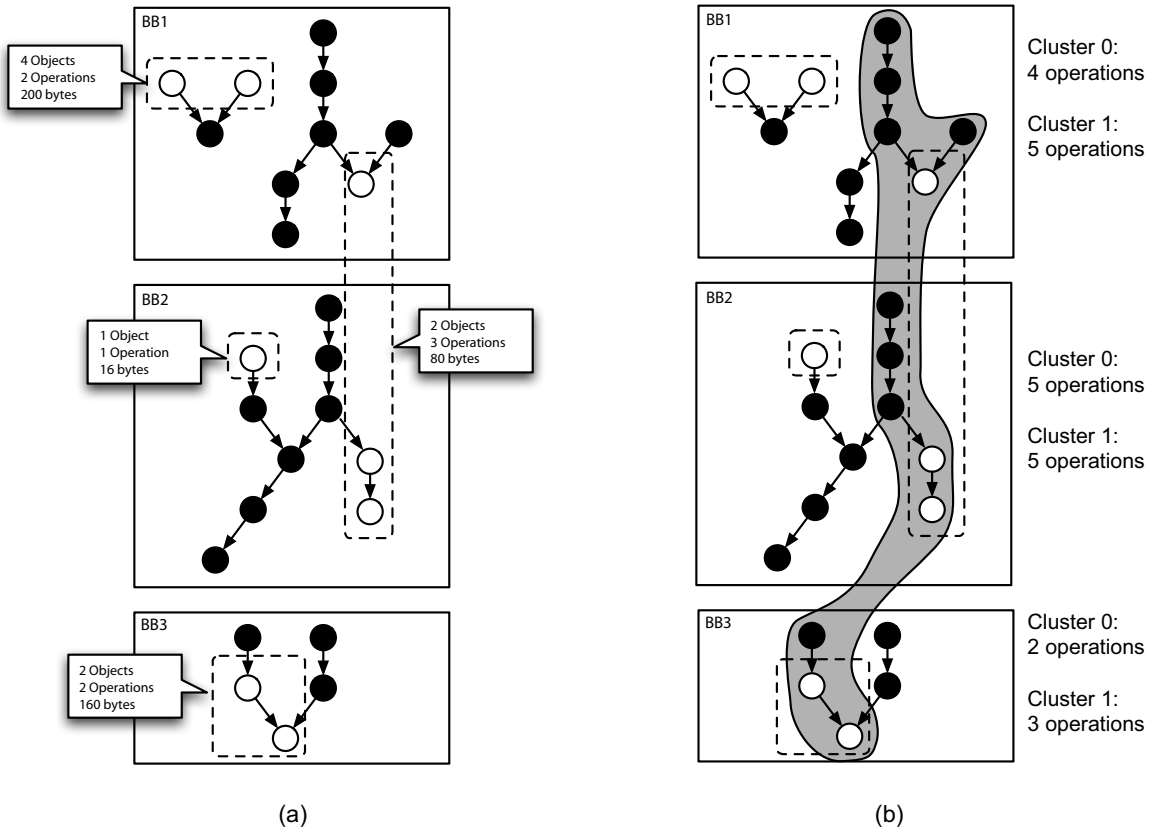

Figure 3.4: Example of global data partitioning with (a) the original graph with operations coarsened with dotted lines and (b) an example good partitioning, indicated by the shaded region.

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 3.4(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.

3.3.5 Second Pass: Region-level Computation Partitioning

The second pass of partitioning uses an enhanced Region-based Hierarchical Partitioning (RHOP) [10], which was introduced in the previous chapter, 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.

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. The RHOP computation partitioning is presented in Chapter 5 and details about how the memory partitioning is taken into account is presented in Chapter 6.

3.4 Experimental Evaluation

This section presents the experiments we ran to evaluate our technique for statically analyzing a program to determine a data partitioning.

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

3.4.1 Methodology

We implemented our experimental framework as part of the Trimaran tool set [67], a retargetable compiler framework for VLIW/EPIC processors. We ran our experiments on Mediabench [44] 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 3.1, and a unified memory model. Table 3.1 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 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 chapter.

Naïve object placement. In the Naïve method, as explained in Section 3.2, 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

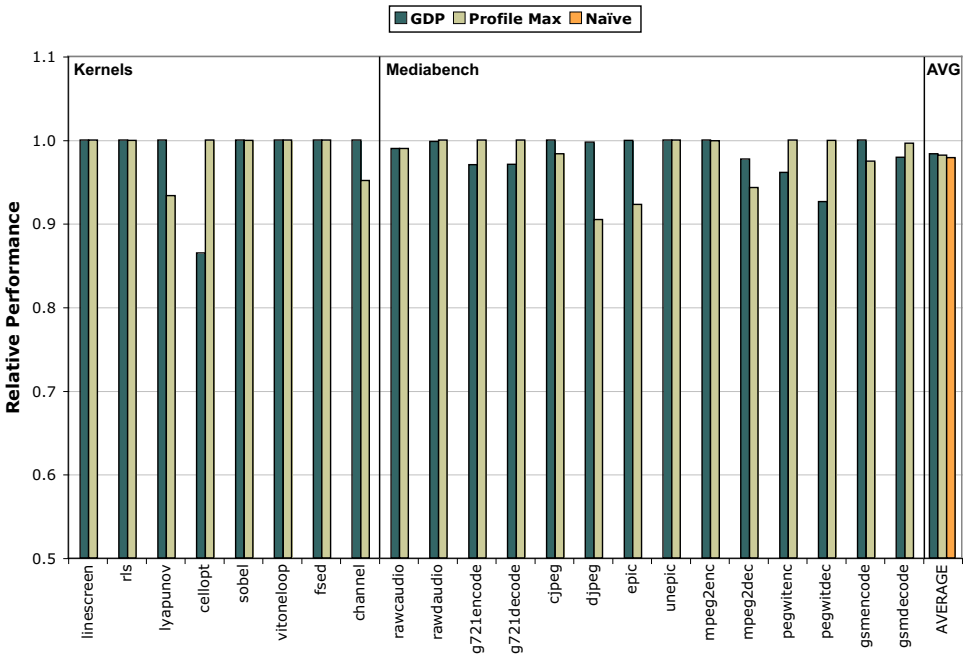

Figure 3.5: Performance of the GDP and Profile Max methods relative to the single, unified memory for a 1 cycle intercluster move latency.

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.

3.4.2 Performance

Figures 3.5, 3.6, and 3.7 show the relative performance of the GDP and Profile Max object partitioning algorithms normalized to the unified memory model with

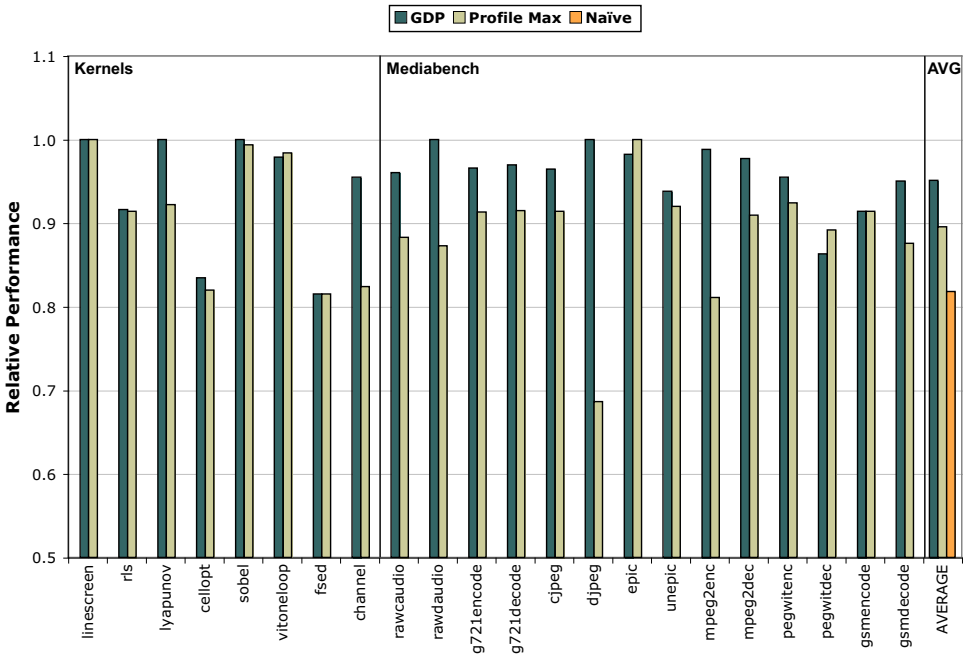

Figure 3.6: Performance of the GDP and Profile Max methods relative to the single, unified memory for a 5 cycle intercluster move latencies.

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 3.1. 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 [10] 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

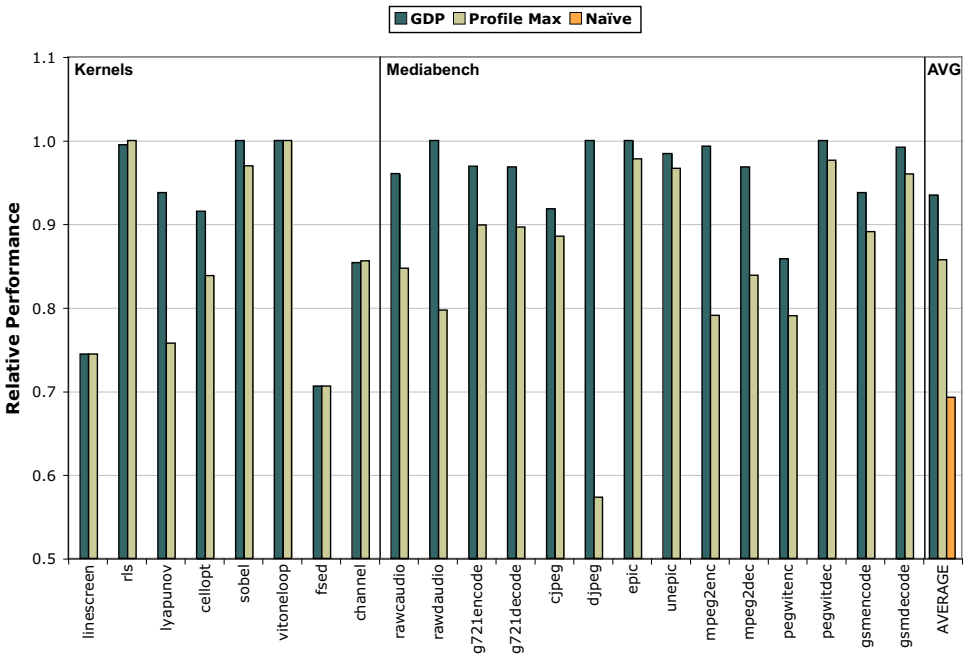

Figure 3.7: Performance of the GDP and Profile Max methods relative to the single, unified memory for a 10 cycle intercluster move latencies.

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 3.5 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 Figures 3.6 and 3.7, 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 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

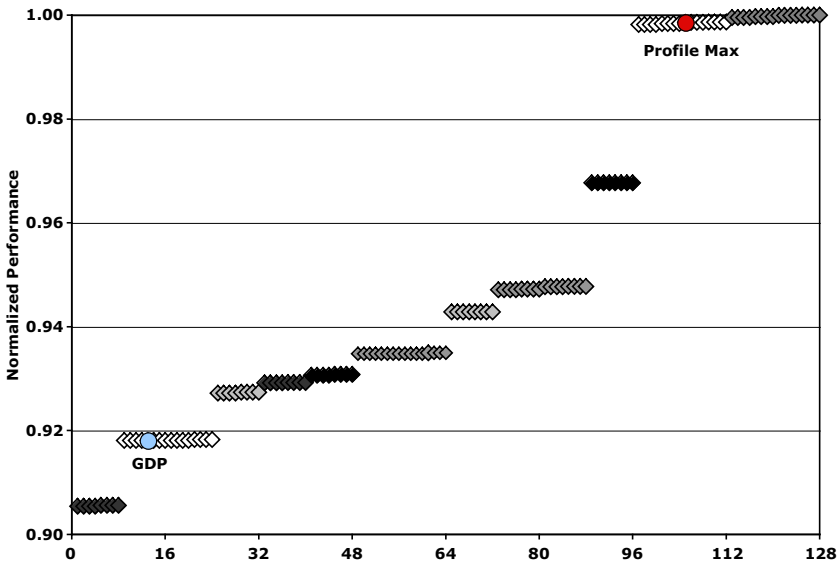

Figure 3.8: An exhaustive search of all possible data object mappings for the rawcaudio benchmark. Points marked with darker shading indicate more imbalanced partitioning in terms of data object sizes per cluster.

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.

3.4.3 Exhaustive Search of Partitions

In Figures 3.8 and 3.9, 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

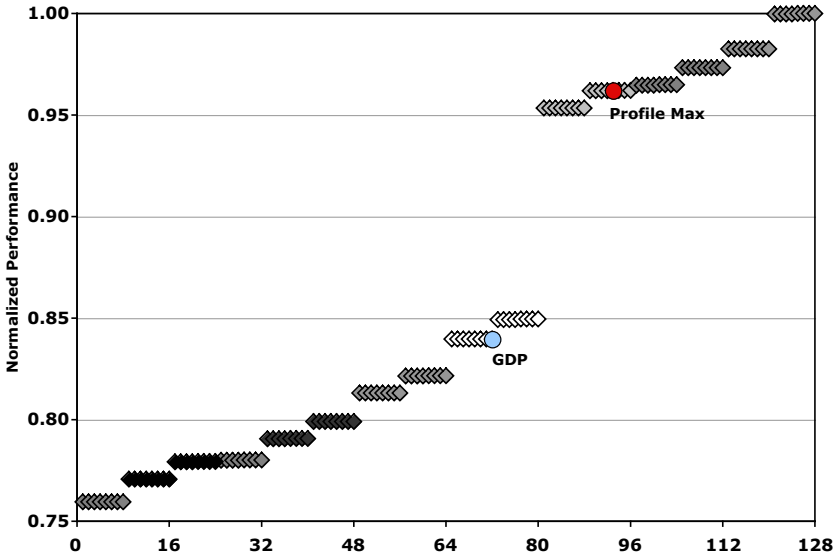

Figure 3.9: An exhaustive search of all possible data object mappings for the rawdaudio benchmark. Points marked with darker shading indicate more imbalanced partitioning in terms of data object sizes per cluster.

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 3.8 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 3.9 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.

3.4.4 Increase in Intercluster Traffic

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

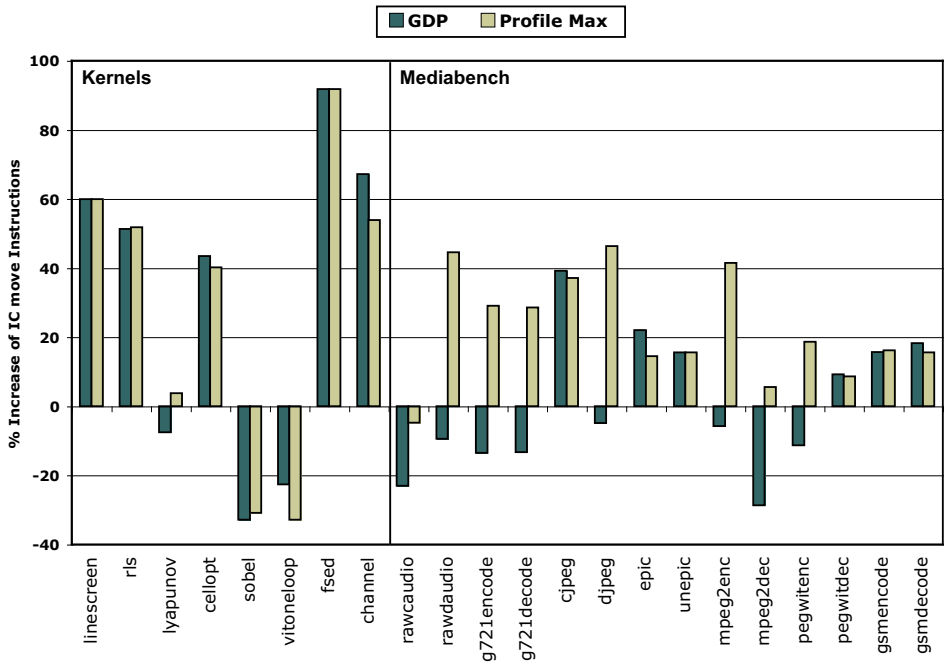

Figure 3.10: The percentage increase of intercluster move operations using the GDP and Profile max methods over a single, unified memory model with 5-cycle latency intercluster move.

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 3.10 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, how-

ever, 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 3.7, as fsed had a large amount of additional moves to insert and had one of the largest decreases in performance.

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 3.7.

3.4.5 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.

3.5 Summary

This chapter introduced 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.

CHAPTER 4

Profile-guided Data Access Partitioning for Distributed Caches

4.1 Introduction

In Chapter 3, a technique for partitioning data objects across multiple distributed memories was presented. While this technique worked well for scratchpad or static local memories, many processors use a distributed L1-cache design, which can allow for a sharing of data values and a more efficient use of the memory. In the presence of data caches, the static analysis technique fails to take advantage the added benefits, as it does not split objects across memories or share the space for frequently utilized data. Statically, it is difficult to determine whether a load or store operation is simply referencing a part of a data object or the object as a whole. This can lead to an unbalanced memory usage, with one cache being issued a large number of access operations because an entire object is assigned to it.

Distributed data caches require the compiler to carefully examine the data access

patterns of each individual memory operation. A good dispersal of data accesses across the cores is critical to producing a high-performance partitioned program. Poor placement of data accesses could lead to significant time stalled waiting for memory because of cache misses or coherence traffic, taking away the gains provided by the fine-grain partitioning of operations. Analysis of the memory accesses of each operation can help to determine when individual data accesses are causing others to either hit or miss in the cache. In addition, the compiler can estimate the contribution each memory operation has to the overall working set. Placing too many operations in a single cache could potentially increase the number of cache misses. Thus, given profile information about affinities between operations and working set sizes, the compiler can proactively combine or split operations across the distributed data caches in order to improve performance.

The underlying vision in this chapter is to compile to chip-multiprocessors, such as RAW [62, 63], that can both exploit TLP and ILP. This chapter focuses on the ILP side, where the architecture can be viewed as a multicluster VLIW with distributed/coherent L1 caches. We propose a new compiler technique that actively partitions memory operations across PEs in order to decrease the memory stall time. A good partitioning of the data allows us to map the problem of fine-grain parallelization for multicore down to the problem of program partitioning for multicluster VLIWs. Our method is a phase-ordered approach to partition memory and computation. Thus, the partition of the data accesses is performed first, regardless of the underlying computation performed. The data access partition is then used the drive the partitioning of the remainder of the code. Our approach first profiles the program

to determine statistics about each memory operation, such as its affinity towards other operations and an estimated working set size. This information is used to create a program-level graph of the memory accesses. The graph is then heuristically partitioned to assign memory operations to PEs. Finally, a detailed partitioning of each code block is performed which respects the preplacement locations of the memory operations.

4.2 Background

This section provides background on our target architecture and the use of a distributed data cache for the memory subsystem. In addition, we introduce some of the challenges faced by the compiler in generating code in the presence of distributed data caches.

4.2.1 Distributed Cache Memories

Both the design choices of using scratchpads or caches for the memory model of a decentralized processor offer several benefits. Scratchpads, which are often used in the embedded design space, have the advantages of being smaller and more power efficient. In addition, scratchpads offer a more simplified model of the memory system for the compiler. Caches can offer increased parallelism from duplicating data objects across clusters, but can suffer from false sharing and congestion in the coherence network. In addition, caches can allow for data objects to have different homes at different times during the execution of the program. A cache also has the potential to be smaller

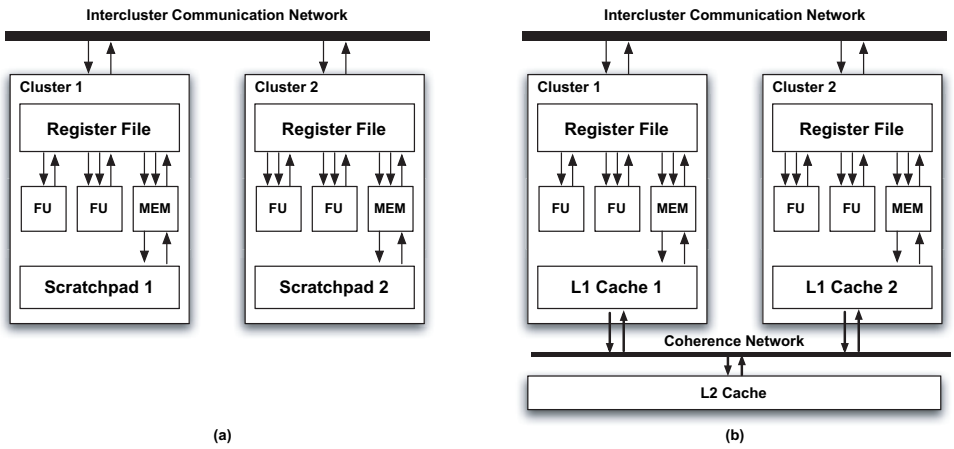

Figure 4.1: Two multicluster processor designs (a) using partitioned scratchpads and (b) partitioned caches

than a scratchpad, since they only need to be big enough to hold the working set of the program, not the entire set of data objects. Figure 4.1 shows the difference between a multicluster processor with partitioned scratchpads and partitioned data memories.

When focusing on partitioned memories for a multicluster architecture, the use of scratchpads can greatly simplify many difficult problems for the compiler. By statically placing an object in a data memory, the compiler will always have knowledge of where exactly its data is located. Thus, when scheduling operations, the compiler can be cognizant of penalties involved with transferring data values across the clusters. The compiler can then easily estimate the required size of a data object when placing it in memory. In addition, given a scratchpad memory size per cluster, it is then simple for the compiler to determine the remaining capacity of the memory. Overfilling the scratchpad memory is not a feasible solution.

Caches require the additional coherence network and must be concerned with the

coherence traffic, as shown in Figure 4.1(b). For a cache-based memory model, data is brought in and flushed out of the cache dynamically at run time. In comparison to the scratchpad model, the actual size of the data object is not as important as the amount each object actually being processed at any given time. Thus, the compiler must account for the time-varying behavior of the object accesses as it is partitioning the data objects. For a cache, rather than total object size, the current working set of a program should remain below the cluster cache size in order to achieve the highest performance.

A second major concern when partitioning the data objects to caches is the utilization of the coherence network. In partitioned scratchpads, data objects are statically placed into their respective clusters at compile time; thus, there is so need for a coherence network or arbitration. The compiler ensures that there will be no contention for data between the clusters. However, with caches, each cluster can have any data object present in its respective cache at any time. The clusters then use a coherence network to arbitrate conflicts between clusters and keep the data consistent. Thus, the coherence traffic generated is yet another factor to consider when partitioning the data objects across clusters.

As our work in [11] showed, a two phased approach to first partition the data objects, then divide the computation cognizant of the data location, can be an effective compiler method for multicluster processors. Our approach makes the partitioning of data objects a first-order term in the mapping of an application to clusters. Thus, the partitioning of the data and how it is accessed is used to drive the later partitioning of the computation operations.

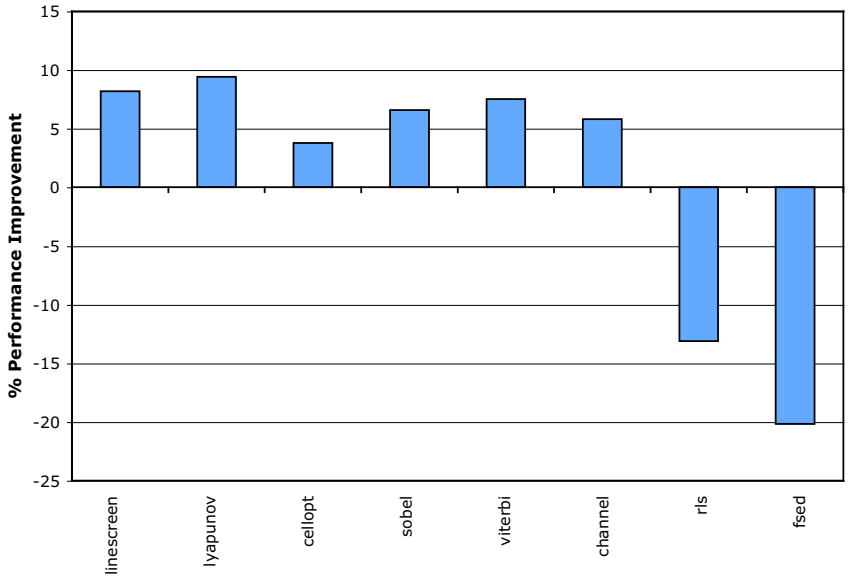

Figure 4.2: Performance of GDP with cache memories.

Figure 4.2 shows the performance of the two-phased Global Data Partitioning (GDP) technique described in Chapter 3, but in the presence of a cache data memory structure. Thus, no modifications were made to the original algorithm to account for the presence of caches. The baseline used for this experiment is a pure run of RHOP as shown in Chapter 5, which assumes a shared memory. GDP mapped the data objects to clusters assuming that they would be placed in scratchpads, but caches were used instead. Each benchmark is shown in comparison to a pure computation partitioning which assumed a single shared memory, but actually had a partitioned data cache. Thus, the baseline was able to bring objects into either cache and use the coherence network to avoid sharing problems.

For several of the benchmarks, the GDP actually improved performance between 5% and 10%. The benefits shown for these mainly result from a decrease in the coherence network traffic. Since the objects and their respective loads and stores

are partitioned across memories, there will be no coherence traffic because there is no sharing of data between clusters. Thus, for coherence traffic-constrained applications, the GDP method of partitioning data objects can already be an effective algorithm to improve performance.

Some benchmarks, like *rls* and *fsed*, show a dramatic decrease in performance. The decrease in these benchmarks stem from the GDP's inability to split or duplicate memory objects across clusters. The GDP models each data object in the program as a single unit, and thus maps each data object to a cluster at an object granularity. Both benchmarks have a large amount of read-only data, but the GDP method is unable to take advantage of sharing the data across the memories. By forcing each of these read-only objects to reside in one cluster, the source data for all the computation can only be driven by a single cluster.

4.2.2 Compilation Challenges for Distributed Data Caches

Producing efficient code for distributed data caches is therefore a challenging task for the compiler. Traditional operation partitioning algorithms [1, 6, 37] assume a shared data memory, which greatly simplifies the compilation task. The compiler could choose to continue use of these algorithms and simply ignore the presence of the distributed caches, allowing the underlying coherence hardware to maintain correctness and properly load objects into the cache when needed. This method passively partitions the memory operations as part of its normal partitioning algorithm, and doesn't consider the underlying hardware. However, this strategy could easily de-

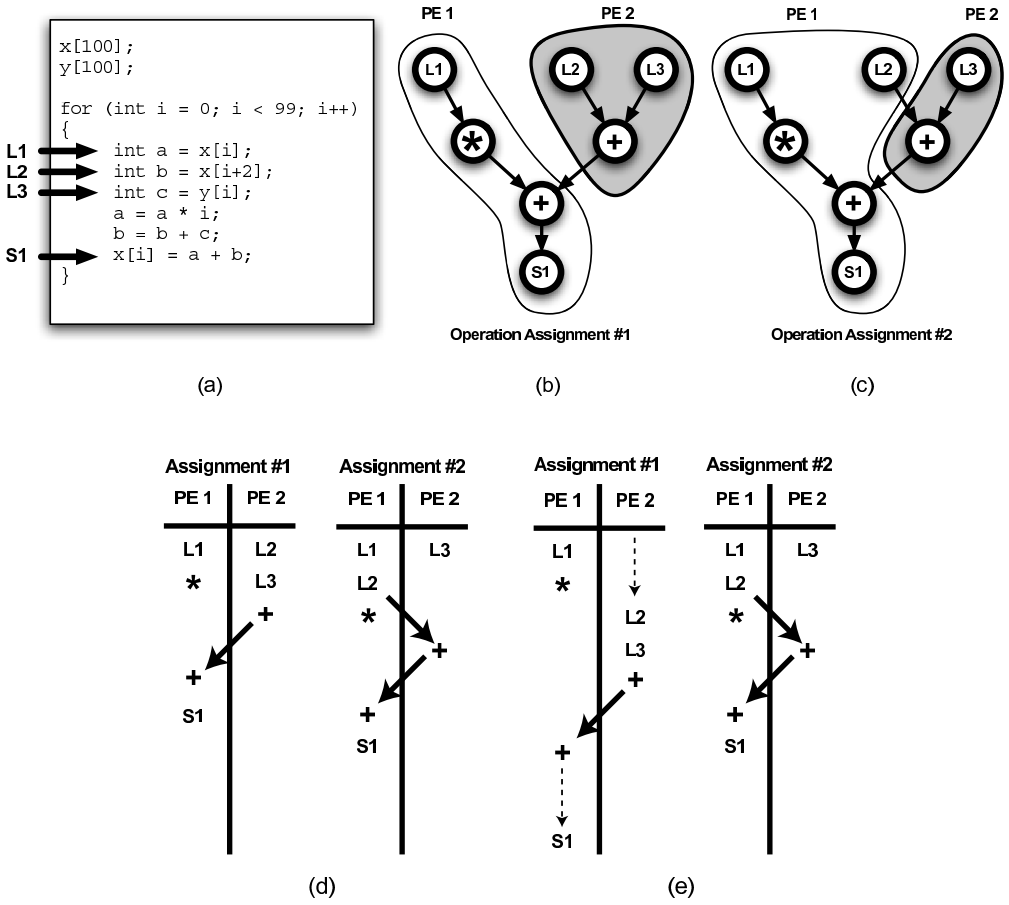

Figure 4.3: An illustrative example of the difficulties in compiling for distributed data caches (a) a code example (b) a partitioning of the operations assuming a shared memory (c) a partitioning of the operations cognizant of the data access pattern (d) idealized schedules assuming a shared memory and (e) schedules factoring in distributed data caches.

grade performance as multiplicative misses, coherence overheads and conflict misses increase. Thus, in order to generate efficient code for distributed data caches, the compiler must actively partition the memory operations cognizant of the cache ramifications of its choices.

Figure 4.3 is an example pseudocode that illustrates some of the difficulties that can arise when the compiler is partitioning the operations. In Figure 4.3(a), the C code for a loop is shown which accesses two different arrays. Within the body of

the loop, there are three loads, two of which are to array `x` and one to array `y`. In addition, there is a store to array `x` at the end of the loop. Each memory operation is annotated with a label, and dataflow graphs for this code are shown in Figure 4.3(b) and (c).

An operation partitioner that assumes a shared memory may try and produce the assignment of operations shown in Figure 4.3(b). This can be a good partition because it only requires one transfer of register values across the communication network, and balances the required work for each PE well. This assignment of operations can place two of the loads (L2 and L3) on one PE and the other memory operations on the other. With a shared data cache between the PEs, the sole objective of operation assignment (including memory operations) is to minimize the expected schedule length.

Given a distributed data cache design, the desired PE assignment can change drastically. Looking again at Figure 4.3(b), in each iteration in PE 1, load L1 will bring a line into the cache that is also written to by store S1. Load L2 is also reading from the same cache line, but in PE 2. When store S1 is executed, its PE will upgrade the line in PE 1's cache to the ownership state, and invalidate the cache line in PE 2. In the next iteration, load L2 will again be executed on the PE 2, causing another miss in its cache, since it had been invalidated. In fact in this next iteration, the miss caused by load L2 will be a case of false sharing of the cache line. It would then have to use the coherence network to get the modified cache line from PE 1. A better partitioning of this code cognizant of the distributed data caches is shown in Figure 4.3(c), where loads L1 and L2, and store S1 are grouped together on a single PE.

The schedules for these two assignments are presented in Figure 4.3(d) and (e). In Figure 4.3(d), we show the idealized schedule with a shared cache. In this case the schedule length of assignment #2 is longer because of the extra required register transfer operations, indicated by the arrows crossing the PE boundaries. Thus, assignment #1 has a shorter per iteration static schedule than assignment #2. However, in Figure 4.3(e), which considers cache effects, load L2 will miss in its cache during each iteration and be stalled (indicated by the dotted line), waiting to transfer the modified cache line from PE 1. In addition, each iteration will have a stall for store S1 waiting to upgrade its cache line to modify it. The schedule for assignment #2 shows none of these coherence issues, and would only stall for cold misses that would affect any partition assignment.

This example illustrates one of the main difficulties in partitioning memory operations for distributed data caches. There is a careful balance between improving cache usage to reduce stall time and the benefits of parallelization. Grouping together all memory operations that access the same addresses onto the same PE can be an attractive option, as it can reduce misses. However, it can also come at the expense of computation parallelism across the PEs. The total execution of the program is the sum of compute cycles and stall cycles, and the compiler must decide which is more beneficial. In this example, it was better to sacrifice computation in order to reduce stall cycles in each iteration of the loop.

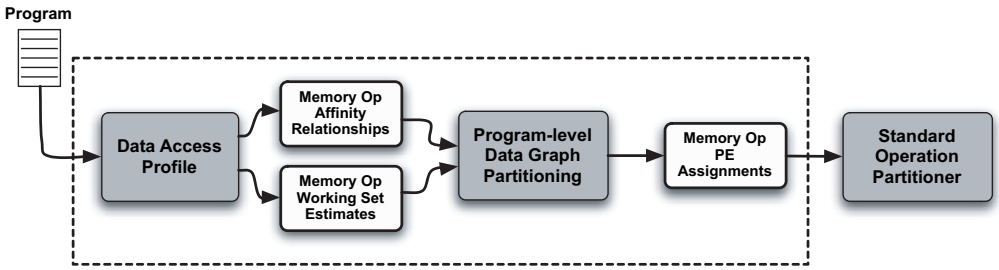

Figure 4.4: A flow chart of our Profile-guided Data Access Partitioning technique.

4.3 Profile-guided Data Access Partitioning

This section introduces our Profile-guided Data Access Partitioning technique for assigning memory access operations to processing elements with distributed data caches.

4.3.1 Overview

This work proposes a profile-guided method that analyzes the access pattern of memory operations and distributes them among PEs. This is followed by a detailed partitioning of the rest of the operations of the program cognizant of the cache location of the memory accesses. We feel this approach to first assign memory operations, then assign the rest of the operations helps to reduce stall-time effects and allows the compiler to take advantage of an improved memory partitioning when generating code for the rest of the program. We chose to use a profile-guided technique rather than static analysis because the profile allows finer grain control over the placement of the data. Static analysis methods can relate memory access operations to data objects, but then the partition must be made at the object level rather than the operation level.

Our technique for data access partitioning includes three steps, shown in the gray boxes in Figure 4.4. The dotted box indicates the main parts of our profile-guided approach. During the first step, a profile of the data accesses is performed which determines affinities that memory operations have with one another. We use a pseudo-cache simulation in order to determine whether any pair of operations would prefer to occur in the same cache or be kept in separate caches. The second step builds a program-level data access graph of all the memory operations, using the statistics gathered during the profile. This graph is then partitioned to determine the memory operation placement across the PEs. In the third step, a standard operation partitioner is used to produce a block-level partitioning of the remainder of the program, cognizant of the preplaced memory operations. This third step is an important part of our process, but is not the focus of this paper. Our main contribution is the development of the profile-guided technique to effectively partition memory operations to reduce stall time. Our technique can be used in conjunction with any standard operation partitioner [11, 16, 56], as long as it can be modified to acknowledge that some of the memory operations have been preassigned to PEs.

4.3.2 Problem Definition

The following definitions formulate the graphs and objectives of our profile-guided data partitioner.

1. **Memory Access Graph:** The graph, $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, where $\mathbf{V}$ is the set of memory access operations (loads and stores), and $\mathbf{E}$ are the edges, which indicate

affinity relationships between the vertices.

2. **Architecture:** Given $\mathbf{k}$, the number of clusters, the machine $\mathbf{M} = \{C_1...C_k\}$, where $\mathbf{C_n}$ is a cluster in the processor with a unique L1 cache..

3. **Partition Assignment:** A function of the form $\delta : \mathbf{V} \rightarrow \mathbf{M}$, wherein each of the memory operations of $\mathbf{V}$ are mapped to a single cluster $\mathbf{C_n}$.

4. **Working Set Size:** A value, $\mathbf{S_n}$, which represents the total size of the memory operations $\mathbf{V_n}$ assigned in memory $\mathbf{M_n}$.

5. **Cut:** An edge, e, is cut if, with respect to the partition assignment δ, its source and destination vertices are mapped to more than one cluster.

6. **Data Partitioning Problem:** Given a DFG $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, find a partition assignment $\delta : \mathbf{V} \rightarrow \mathbf{M}$ that maps the objects of $\mathbf{V}$ onto one of the $\mathbf{k}$ disjoint caches such as the weight of the cut edges is minimized and the working size, $\mathbf{S_n}$, across clusters is balanced.

4.3.3 Data Access Profile

The first step of our technique is a profiling pass that builds a program-level data access graph between memory operations. The purpose of this step is to monitor the data accesses produced by each memory operation to determine whether individual operations have any preference to being placed together on the same cache or apart on different caches. Thus, the resulting data access graph consists of each memory operation in the program connected by edges representing the affinity between memory

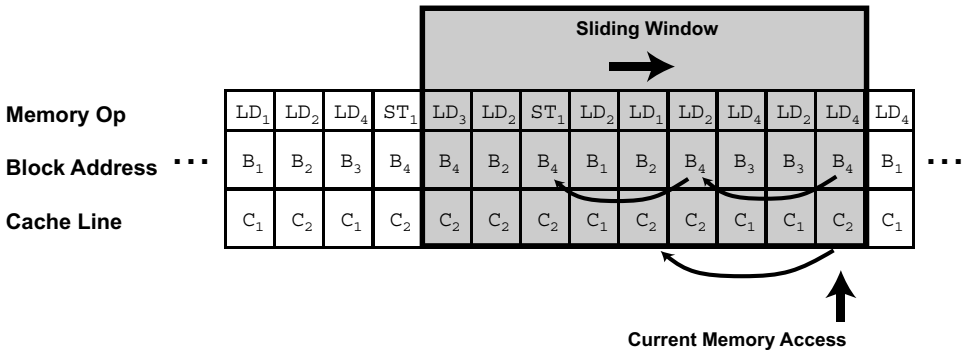

Figure 4.5: Our profiler's sliding window to analyze memory access affinities

operations.

We identified three important characteristics of memory operations to take into account during this pass: *constructive interference*, *destructive interference*, and *working set size*. Constructive interference occurs when two memory operations are likely to access the same cache lines and one access helps the other hit in the cache. Similarly, destructive interference occurs when two memory operations are likely to kick one another out of the cache. Working set size is an estimate made of how much memory an individual memory operation takes up in the cache. Each of these three characteristics is determined during a profile run by examining and analyzing the stream of memory accesses produced by the program.

Our data access profile creates a sliding window of the last n memory accesses, as shown in Figure 4.5. The sliding window gives the profiler a narrow view of the recent accesses that have been sent to the data cache. With each access, the profiler records the memory operation and the accessed address into the window. The operation and address are recorded in order to determine if future memory accesses can benefit from being in the same partition as the current access. Thus, the profiler is performing

a pseudo-cache simulation over the window of accesses, trying to determine whether memory operations attract or repel one another.

The profiler then travels back through the window of accesses from the head to the tail, recording a count of each time it sees a different memory operation that accessed the same memory block. This static count of accesses to the same address is an indication of how much the memory operation could help the current access hit in the cache. Thus, this value is used as the constructive interference metric, or positive affinity of operations to one another. For the case in Figure 4.5, our profile statistics would increase the constructive interference between LD 4 and LD 2 once as well as LD 4 and ST 1 because they all access block B4 within the window. Equation 4.1 illustrates the calculation for the constructive interference between a static memory operation i and another operation j.

$$cons(i,j) = \sum_{trace} \sum_{window} (line(i) == line(j)) \tag{4.1}$$

While traveling back through the window of accesses, the profiler also records how often the current memory access is likely to kick another memory operation's accessed cache line out of the cache assuming a direct-mapped cache. This static count is used as the destructive interference, or negative affinity of operations to one another. In Figure 4.5, since blocks B4 and B2 both write to cache line C2, the profile increases the destructive interference between LD 4 and LD 1 by one. In both the constructive and destructive passes back through the sliding window, the analysis is terminated either at the tail of the window or when an address match occurs with a memory

```
Input: {(L_1, B_1), (L_2, B_2), (L_3, B_3), ..., (L_k, B_k)}
Output: {L_k, Size}
for (i = (k − 1) to 1) do
    dist++ ;
    if (B_i = B_k) then
        break;
    end
end
 if (DistMap[L_k].lookup(dist)) then
    DistMap[L_k].value(dist)++ ;
end
else   DistMap[L_k].insert(dist, 1) ;
```

Algorithm 1: **Estimating working-set size of load/store instructions.**

store operation. The store operation termination was added because with coherent L1 caches, a store would cause an exclusive access in one of the caches, effectively invalidating it from the others. Thus, in Figure 4.5, the profiler would begin at the current memory access LD 4, and travel back noting that load LD 2 and store ST 1 accessed the same block. However, it would not mark that load LD 3, the tail of the sliding window, also accessed that block, since the store ST 1 would have ownership on that cache line. The calculation for destructive interference between a static memory operation i and another operation j is this illustrated in Equation 4.2.

$$dest(i,j) = \sum_{trace} \sum_{window} (block(i) == block(j)) \tag{4.2}$$

The final metric recorded by the profiler is an estimate for the working set contribution of each memory operation. For this work, we leverage previous research for working set estimates based on reuse distances of load and store operations [4, 53]. Algorithm 1 illustrates the calculation of this estimate. Input into the algorithm is a list of cache block references for every load and store operation in reference order.

The working set size is estimated by looking at past references to unique blocks by that instruction from the head to the tail. The algorithm measures the distance, or number of intervening operations between the current instruction and its accessed block and its previous instance. This quantifies the number of cache blocks that must be present in a fully-associative cache for the instruction to result in a hit. For each memory operation, a histogram is produced that indicates the number of times the operation had each access distance. A weighted average is taken of these cache distances and is used as the working set estimate. Thus, while this estimate is not perfect as it assumes a fully-associative cache, it can be useful as a heuristic to help determine how long certain accesses would likely prefer to be in the cache.

Our memory partition method uses a direct-mapped cache metric for our interference calculations and a fully-associative cache metric for the working set estimates. While these are very different cache designs, using them for the metric allows for a more conservative estimate. The direct-mapped assumption for the interference estimates the worst case situation for conflict misses in the cache. Similarly, the fully-associative assumption estimates the worst case for capacity misses. Thus, these assumptions can allow for a more conservative view of the cache effects of the memory operations.

4.3.4 Access Partitioning

The second step is the actual partitioning of the data accesses across the different data caches. After the profile run is completed, a program-level graph of all the

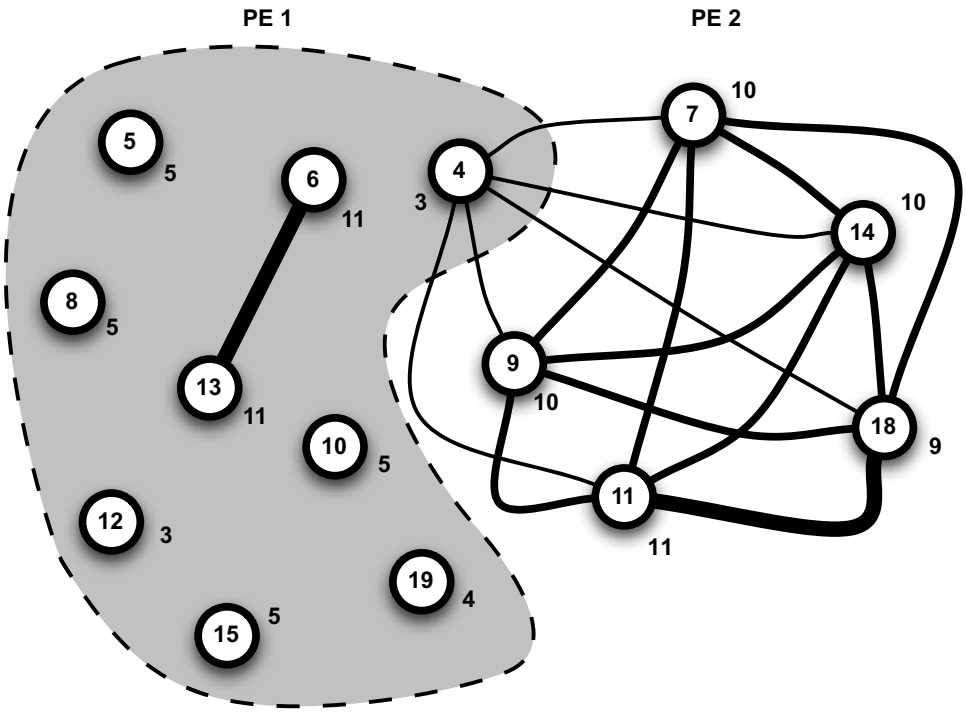

Figure 4.6: Partitioning of the program-level data access graph to processing elements for the *rawdaudio* benchmark. Nodes represent memory operations are annotated with their working set estimate. The thickness of the edges indicate the amount of affinity one operation has for another.

accesses in the program is created. The nodes in the graph represent each memory operation and the edges connecting the nodes represent affinities between the operations. Each node is weighted with the estimated working set size of the memory operation. Each edge is weighted with the difference between of the constructive and destructive interferences between the memory operations. Our technique inputs this graph into the METIS graph partitioner [38], which divides the data accesses trying to cut as few positive edges as possible while keeping a balanced node weight per partition.

The use of the estimated working set for the node weight helps keep the partitioned memory operations balanced in their usage of the data caches. For example, if a

memory operation had a high working set estimate, it would most likely need to be in the cache for a longer period of time to hit than an operation with a low working set estimate. Thus, the partitioner could take this into consideration and not push as many large working set operations together into the same cache.

The use of the constructive and destructive affinities for the edge weights helps the partitioner decide which edges are best to cut when dividing up the memory operations. If two operations had a large amount of destructive interference, this would be indicated in the graph with a large negative edge weight. That would then be attractive to the partitioner as a good edge to cut. Similarly, if two operations had a large amount of constructive interference, the edge weight would be a large positive value, indicating the operations should be kept together in a single cache.

Figure 4.6 shows the data access graph created for the *rawdaudio* benchmark. In this figure, each node represents a memory operation and is labeled with an id number inside and its estimated working set next to the node. Edges connect various operations in this graph, and the thickness of the graph indicates different affinities. Higher affinities are shown with thicker edges. Thus, operations 11 and 18 have a very high amount of affinity towards one another, while 4 and 7 have a very small amount. Nodes without any edges are memory operations that had no affinity towards any other operations during the profile. For this benchmark, there was no destructive interference, as is common with many kernel loop benchmarks that generally walk through large data arrays. Therefore, there is no negative affinities for the partitioner to try and push operation apart, but the partitioner still had to deal with which operations to merge together and how to balance the working set.

The end partition in Figure 4.6 shows several interesting results. First, the most highly connected nodes, 6 to 13 and 11 to 18, are kept together on the same PE. This allows them to help one another in using the same cache lines. In addition, the partitioner chose to group many of the other operations with affinity to 11 and 18 together. Next, many of the operations (5, 8, 10, 12, 15 and 19) have no edges, suggesting they have no interaction with one another. In these cases, the partitioner chose the group them with operations 6 and 13 to help balance the overall working set sizes. The large group of operations (4, 9, 11, 14 and 18) that had high affinity towards one another were grouped together and they had a large working set sum, thus the partitioner preferred to place the free operations on the other PE. Finally, note that the partitioner chose to cut the edges around operation 4. This was done to also help balance the working set, as all the operations left in PE 2 have a large working set size. Operation 4 was chosen because it had the smallest affinity towards the other operations and created the best balance of the working set.

4.3.5 Operation Assignment

The final step of our technique is to finish partitioning the rest of the program code, including all computation operations. In this step, any standard operation partitioning technique can be used, as long as it can be modified to be cognizant of preplaced memory operations. For this work, we used the Region-based Hierarchical Partitioning (RHOP) algorithm [10] to distribute all the operations across the PEs. RHOP is a operation partitioner capable of efficiently generating high-quality oper-

ation divisions; however, as with most previous operation partitioning algorithms, 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 complex 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 PEs 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. A more detailed explanation of the RHOP algorithm is provided in Chapter 5.

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. Our process is illustrated in Figure 4.7 for a basic block in the benchmark *rawdaudio*. In Figure 4.7(a), the partitioning of the block begins with the memory operations

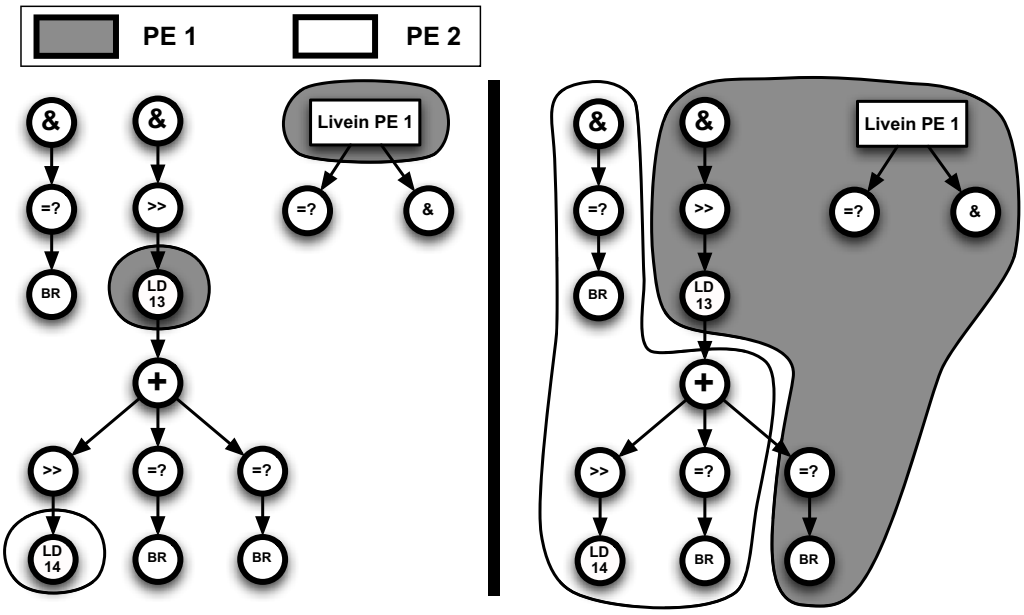

Figure 4.7: Operation assignment of the computation operations in *rawdaudio* given the memory operation placement from our profile guided technique.

preassigned to their respective PEs from the output of the previous step (as shown in Figure 4.6). Every other operation is unassigned to a PE, but some operations have live-in operands from previously partitioned blocks in the code. In Figure 4.7(b), we show the operation partition after RHOP has completed. The memory operations are still assigned to their prebound PEs, and the operations around them have been partitioned taking the data location into consideration. There are two transfers across the communication network in this figure, one from LD 13 to an add operation, and one from the add operation to a compare operation.

While RHOP can produce high-quality partitions of the operations, it has the underlying assumption that data can be accessed from any PE. We extended RHOP to allow for a prebinding of memory operations to the PE which was determined by the partition of the program-level data access graph. This was done by modifying the

resource usage weights of the memory operations to have a very high weight when placed on an incorrect PE. Furthermore, operations prebound to different PEs are prevented from coarsening with one another. When RHOP begins estimating costs of moving computation operations between partitions, it can then take into account the moves that may be required to transfer the accessed data across PEs.

4.4 Experiments

This section presents the experiments we ran to evaluate our profile-based method for data access partitioning.

4.4.1 Methodology

Our profile-guided data access partitioning technique was implemented as part of the Trimaran compiler infrastructure [67], a retargetable compiler for VLIW/EPIC processors. The machine model used had 2 or 4 PEs and 1 integer, float, memory and branch unit per PE. Each PE includes a distributed L1 data cache of varying sizes between 512B and 8kB. We assumed a shared 128kB 4-way associative L2 data cache and coherency was kept between the L1 caches with a MOESI coherence protocol. The intercluster communication network between PEs, which is used to transfer register values, allows for a total of 1 move per cycle with a 1-cycle latency. More details of our simulated machine are provided in Table 4.1.

We ran our experiments on a wide number of benchmarks with varying amounts of inherent parallelism in order to gauge the effectiveness of our technique. The

Parameter	Configuration
Number OF PEs	2, 4
Function Units	1 I,F,M,B per PE
PE Comm. B/W	1 total move per cycle
PE Comm. Latency	1, 2, 3 cycles
L1 Cache	2-way associative
L1 Block Size	32 bytes
L1 Cache Sizes	512B, 1kB, 4kB, 8kB per PE
L1 Hit Latency	1 cycle
L1 Bus Latency	2 cycles
L2 Hit Latency	10 cycles
Main Memory Latency	100 cycles
Coherence Protocol	MOESI

Table 4.1: Details of our simulated machine configurations

benchmarks with the most parallelism consisted of a set of DSP kernels. These benchmarks were the ideal case with a significant amount of available parallelism to extract. We also used the Mediabench [44] benchmarks, which have slightly less parallelism, but still enough to extract some performance gains. Finally, we ran our technique on the SPEC CPU 2000 benchmarks, which have the lowest amount of parallelism available. Our results show a representative set from these benchmark suites.

For each benchmark, we evaluated the performance of a standard RHOP generated partition which assumes a shared memory and compared it to our profiled-guided method which prepartitions the memory accesses. Thus, our base case is a data cache incognizant method, which places data access operations without knowledge of the distributed L1 caches. We report the improvement of our technique relative to the base RHOP case. A comparison with the base RHOP partitioning was used to determine the amount of improvement that our phased-ordered memory placement

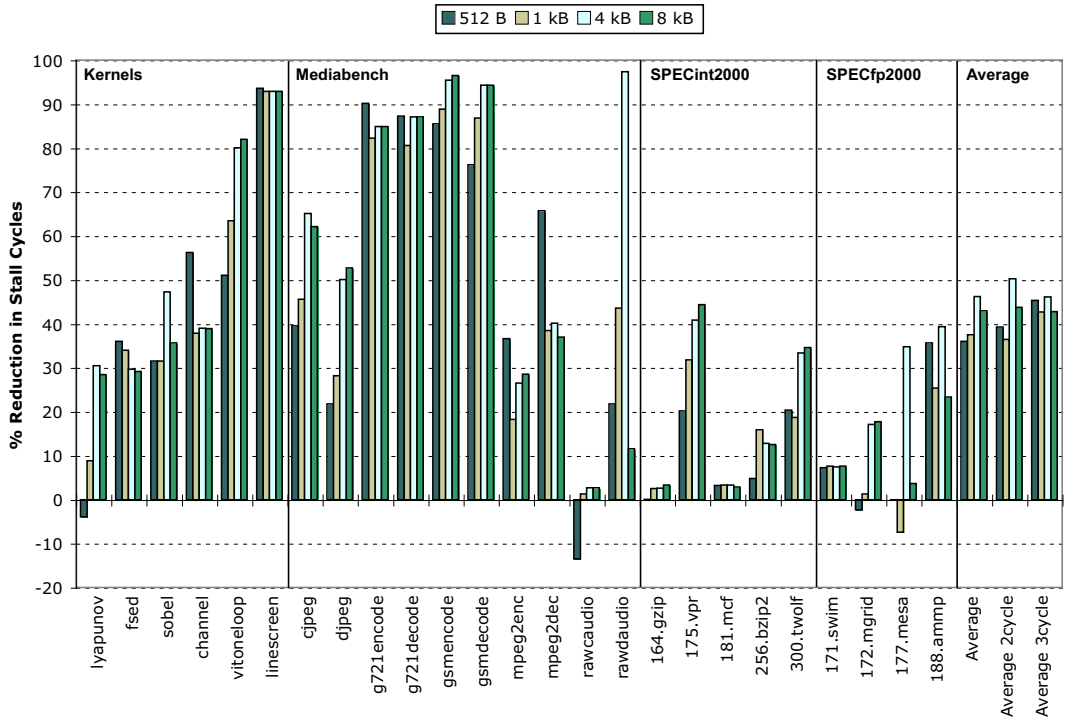

Figure 4.8: Reduction in stall cycles when using a profile-guided data access partitioning for a 2-PE processor

could have on a standard multicluster operation partitioner. In generating our PE assignment for memory operations, our data access partitioning technique profiled each application using a sliding window size of 256 instructions and assumed a 32-byte line size. Each benchmark was profiled and evaluated on different input sets. The profile used a smaller input set to generate the memory operation to PE bindings.

4.4.2 Performance Improvement

Figure 4.8 shows the improvement in stall cycles for our profile-guided data access partitioning technique compared to the partition produced by RHOP with no active data partitioning. Each bar represents a different cache size per PE. Higher bars indicate a larger reduction in stall cycles, and bars below zero indicate a increase in

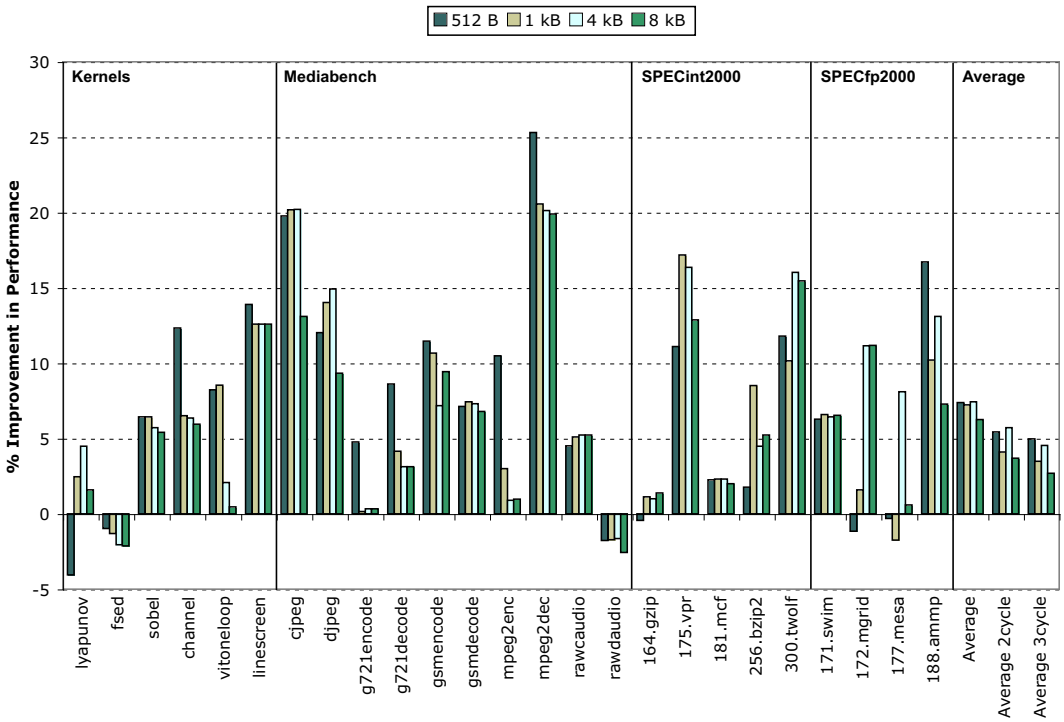

Figure 4.9: Total performance improvement of our data partitioning technique.

stall cycles. In almost all cases our technique significantly reduced the number of stall cycles, as much as 90% in *gsmdecode* and *linescreen*. This can be attributed to a better grouping of high affinity memory operations decreasing the coherence traffic and better localizing data usage in a single PE. Most benchmarks saw increasing benefits as cache size increased, as a larger cache size allowed for our grouping of memory operations together to be more effective at keeping cache lines valid. Some, like *rawdaudio*, showed drop-offs in improvement at the larger cache sizes. This was not due to the profile-guided approach performing worse, but attributed to the fact that almost all of the working set of the benchmark could fit in the larger cache.

In addition, the average stall cycle reductions for intercluster communication latencies of 2 and 3 cycles are shown. For higher latencies, the stall cycle reduction is

fairly similar. Variance appears because RHOP reassigns the computation causing slightly different overlaps between computation and stall time.

Some benchmarks, like *lyapunov* and *rawcaudio*, actually increased the number of stall cycles with a 512B cache. For *rawcaudio*, this occurred because the partition assignment overcommitted one of the PE caches with too many memory requests, causing additional conflict misses. This resulted in this benchmark making far more accesses to the L2 cache than the base case. For the *lyapunov* kernel, our partitioning of the memory operations actually produced a memory operation assignment that had comparable L1 cache miss rates. However, with the small cache size, our partitioner overcommitted the number of store accesses assigned in PE 2, increasing the number of misses in one critical section.

While the number of cycles due to stall is improved, total performance is affected by the sum of both stall and compute cycles. Figure 4.9 shows the results for the overall performance factoring in compute cycles. Again, higher bars indicate an increase in performance, and bars below zero indicate a performance loss. Overall, most benchmarks show performance improvement, although it is not quite as pronounced as the improvements in stall cycle time. Some benchmarks seemed to perform better with larger cache sizes. This occurred because our method of grouping together high affinity memory operations could more likely keep their accessed data in the larger cache. In addition, with increased cache sizes, the cache-unaware base RHOP partitioner would more likely have lines sitting in its cache that would need coherence arbitration to invalidate.

In many of the benchmarks, compute time was a much larger portion of the total

the execution time than the memory stall time. Since we are not actively addressing the compute benefits, this decreased the benefit of improving stall cycle time. In the cases where performance worsened, our phase ordering of the memory assignment often restricted the parallelism that the computation partitioner could extract. Again, we show the average total improvement in performance for 2 and 3 cycle intercluster communication latencies. While the stall cycle reduction remains quite similar, as shown in Figure 4.8, the total performance improvement decreases with higher move latencies. This is because the higher latency intercluster communication causes an increase in the computation cycles, thus making the stall cycle improvements a smaller fraction of the total cycle time. Our profile-guided method can only actively address stall cycle improvements. However, we still achieve an average performance improvement of 6-7% for each cache size with a 1-cycle communication. In some cases, like *cjpeg* and *mpeg2decode*, the performance improvement is more than 20%.

4.4.3 Reduction in Coherence Traffic

Besides stall cycle reductions, another benefit of our data access partitioning method is that by actively grouping memory operations that have high affinity towards one another, coherence traffic can be reduced. While this is not a strict indicator of performance, if the coherence traffic became a bottleneck, this reduction could be beneficial. With a significant increase in network traffic, congestion could cause additional delays on memory accesses, as coherence requests begin lining up waiting for arbitration.

Benchmark	Incognizant	Profiled	% Reduction
lyapunov	82117	0	100.0
fsed	335597	8253	97.5
sobel	841	69	91.8
channel	3196	179	94.4
vitoneloop	16834	1	99.9
linescreen	1416908	3	99.9

Kernels

Benchmark	Incognizant	Profiled	% Reduction
cjpeg	525562	52021	90.1
djpeg	160703	7930	95.0
g721encode	9384112	672792	92.8
g721decode	9353752	737476	92.1
gsmencode	4758854	194416	95.9
gsmdecode	3082861	275893	91.0
mpeg2enc	4138884	1213889	70.7
mpeg2dec	12956935	1916597	85.2
rawcaudio	76	4	94.7
rawdaudio	102580	1	99.9

Mediabench

Table 4.2: The number of snoops required across the coherence network for the pure RHOP case and the Profile-guided case for high ILP kernels and Mediabench, as well as the percentage reduction in snoops.

Tables 4.2 and 4.3 presents the reduction in coherence traffic produced by our profiled-guided method for a 4kB data cache per PE. For each benchmark, the number of snoops performed by the base RHOP and the profile-guided method is shown along with the percentage reduction. In almost all cases, our method is able to significantly reduce the number of snoop requests put on the coherence bus. Most cases are above an 80% reduction in snoop requests, with the lowest being a 47% reduction in *mpeg2enc*.

There was only one benchmark, *177.mesa*, in which our technique failed to significantly reduce the coherence traffic. This application had a small region with a large number of stores to similar locations. Our technique chose to try and balance out

Benchmark	Incognizant	Profiled	% Reduction
164.gzip	137074797	9795008	92.9
175.vpr	11192509	1711914	84.7
181.mcf	38087128	11278386	70.4
256.bzip2	150271055	5198114	96.5
300.twolf	49666748	2147275	95.7

SPECint

Benchmark	Incognizant	Profiled	% Reduction
171.swim	4500483	5264423	64.3
172.mgrid	2649332	9795008	94.8
177.mesa	31104748	30271083	2.7
188.ammp	1024210	273371	73.3

SPECfp

Table 4.3: The number of snoops required across the coherence network for the pure RHOP case and the Profile-guided case for low-ILP SPECcpu benchmarks, as well as the percentage reduction in snoops.

the memory usage and caused the majority of the coherence traffic. Another benchmark to note is *rawcaudio*, which had a extremely small number of snoops in the base RHOP case. This indicates that the base RHOP had a fairly good partitioning of the data objects, as it didn't require many invalidations. Figure 4.8 shows that we were not able to significantly improve this benchmark. This table shows that the base case was already producing a fairly good result; thus, there was little room for improvement.

4.4.4 Partitioning to Four Processing Elements

To see how our algorithm generalizes to more parallel architectures that are becoming more common today, we ran our method on a machine with 4 PEs. Figures 4.10 and 6.3 presents the percentage reduction in both stall cycles and overall speedup for two different machines: a 2-PE processor and a 4-PE processor. In each case, each

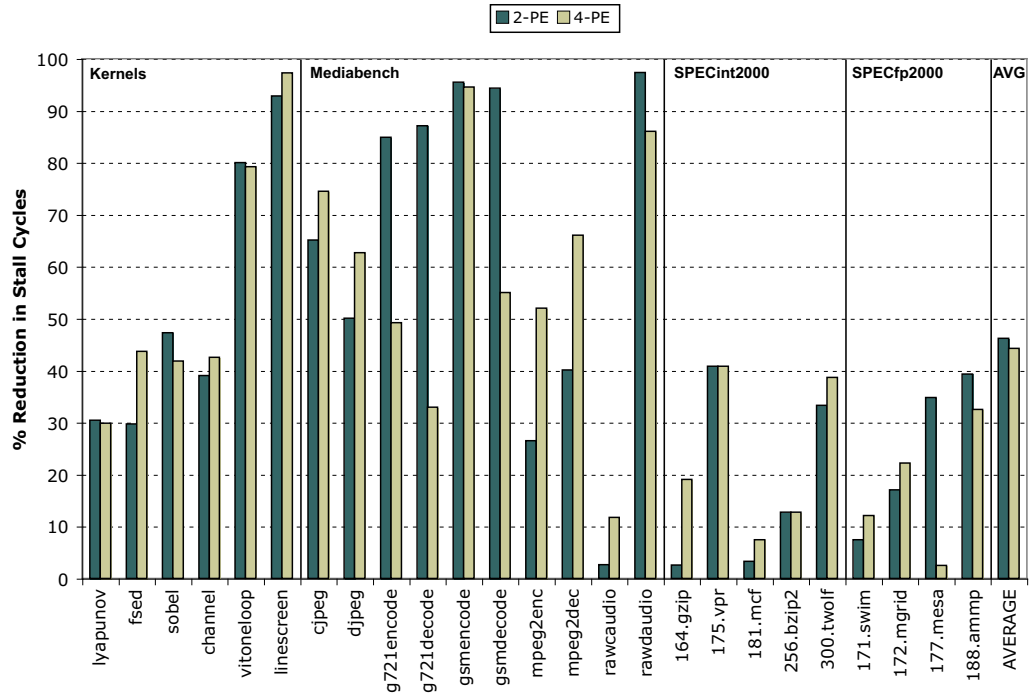

Figure 4.10: Comparison of 2-PE and 4-PE machines for stall cycle reductions

PE had a 4kB L1 cache associated with it. In Figure 4.10, the two bars indicate the percentage reduction in stall cycle time for the 2-PE and 4-PE machines, respectively. In Figure 6.3, the two bars indicate the overall speedup achieved over a single PE machine for 2-PEs and 4-PEs. In each case, higher bars indicate better performance.

In general, the results of the 4-PE processor were mixed, but overall they were fairly in line with the results for 2-PE processors. The stall cycle reductions in Figure 4.10 indicate that with more PEs, our proactive technique was still able to reduce a large amount of the memory stalls. In some cases, this reduction was significantly larger, such as *mpeg2enc*, in others, it was less such as *g721encode*. However, on average, we were able to reduce approximately the same amount of memory stall cycles.

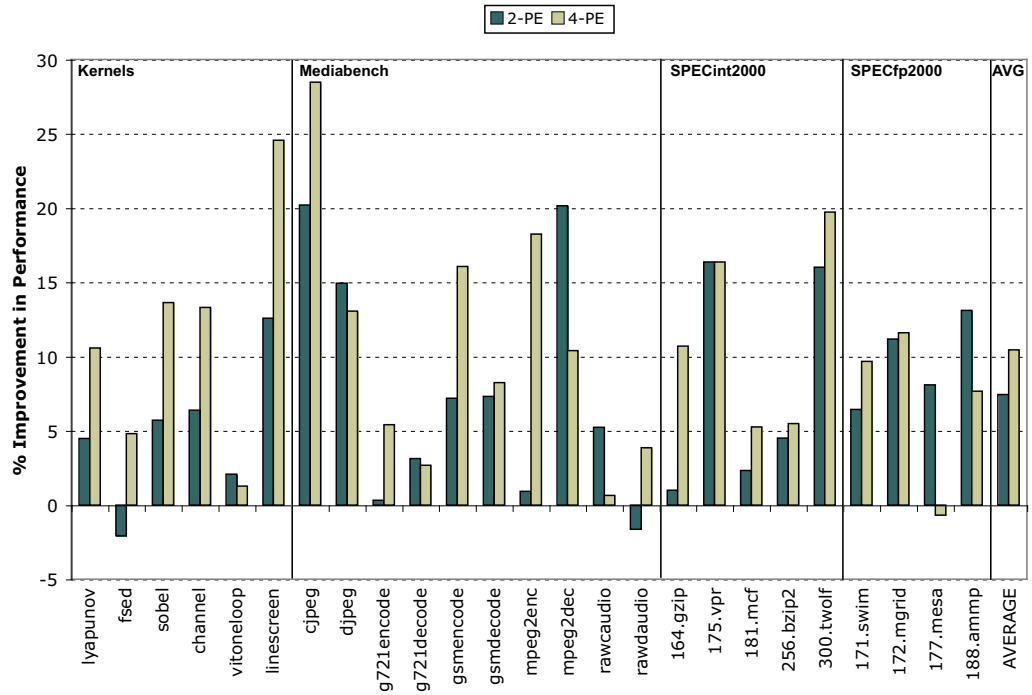

Figure 4.11: Comparison of 2-PE and 4-PE machines for overall performance

Figure 4.11 shows that overall performance was also similar to our 2-PE results. Some benchmarks, such as *cjpeg* and *gsmencode*, where dramatically improved; however, on average, there wasn't a significant increase in performance even though were four times the number of resources as the baseline case. Much of the achievable performance benefits depends on whether or not the benchmark has enough parallelism to support the wider machine. In addition, the larger number of PEs increases the contention for the communication bus causing more compute cycles to be executed. This shows us that while fine-grain parallelism is useful and can be exploited for performance improvement, it has its limits based on the application.

4.4.5 Compile-Time Effects

Our method for profile-guided memory operation assignment will generally increase compilation time. The runtime of the profiler is affected by three major factors: the profiling input used, the size of our sliding window, and the total number of memory operations executed by the program. For our experiments we used a smaller input to profile all the benchmarks. The sliding window size was kept at 256 instructions, so every executed memory instruction would at most look at the last 256 traced memory addresses. Our current system is not optimized for speed and can be improved significantly. Overall runtimes varied by benchmark; the worst case was approximately doubling of the total compile time. These cases were generally benchmarks which had many memory instructions to trace. On average, our technique increased compile time by 30.8%.

4.5 Related Work

The topic of compiler partitioning for distributed architectures has been studied significantly in the past, especially in the context of multicluster VLIW processors. The first cluster assignment algorithm for multicluster processors was the Bottom-Up Greedy (BUG) algorithm in the Bulldog compiler [16]. BUG greedily assigns operations to clusters in order to minimize a estimated schedule length. Thus, it had a very narrow view of the code and often fell into local minima. Özer et al. [56] developed a partitioning algorithm that unified cluster assignment with scheduling to produce more efficient code. Many other operation partitioning techniques [1, 6, 37]

have been proposed, but all make the fundamental assumption of a shared data cache to reduce the complexity of their algorithms. We view our profile-guided data access partitioning work as an additional prepass phase which can work in concert with any of these algorithms to help produce better partitions in the presence of distributed data caches.

Recently, there has been related work in the area of partitioning data objects across distributed data memories. Sánchez [59, 60] studied the fully distributed clustered VLIW architecture, where fetch, execute and memory units were all decentralized. They focused on modifying the modulo scheduler to be cognizant of memory locations. Their technique similarly uses a locality metric; however, they focus on cache misses. In addition, their cache miss equations, which determine the number of misses caused by memory instructions, work on affine arrays. Our technique is more global in nature, as it determines both positive and negative affinities throughout the entire program and can handle arbitrary code. Gibert et al. [26] use small low latency buffers as localized storage and dynamically fill them in order to improve performance. Other recent work by Gibert et al. [24, 27] partitions memory at a data object level into either a fast-access, high-power cache or a slow-access, low-power cache. Thus, objects that are accessed in critical portions of the code are placed into the fast cache, while non-critical objects can be partitioned into the slower cache. Their work is not directly comparable to ours, as they target power reduction rather than performance. Hunter [36] investigated placing objects into specialized SRAM arrays, and focused on lowering the memory port requirements and access latencies. Chu et al. [11] studied the partitioning of objects into scratchpad memories. Their work considers data and

an object level and partitions entire objects across the memories. This allowed for simplification of the partitioning problem, as the compiler could ignore the effects of coherence and sharing of data. However, their technique was unable to consider working set sizes and were forced to use the larger granularity of an object rather than individual loads/stores.

In the multiprocessor domain, there has been previous work on software mechanisms to reduce the required invalidation and updates from coherence traffic. Cheong et al. [9] proposed a method to selectively invalidate potentially stale cache lines, thus proactively reducing the required invalidations. Chen et al. [8], developed a compiler method to allow for writes to shared cache lines, which would also reduce the required coherence traffic. These works focused mainly on individual accesses and reducing specific coherence request instances, rather than the producing an overall partition of the program.

Overall, we see our work having several advantages and improvements over previous memory partitioning techniques. First, rather than using a data object granularity for partitioning memory, we use a much finer-grain memory operation granularity. Thus, rather than partitioning an entire object to a cache, we can partition individual load and store operations that may access different portions of the data object to different caches. Second, we use statistics gathered in a memory access profile to help guide the partitioner to group and repel operations with one another. Third, we break the assumption of static scratchpads and take the use of data caches into consideration. Lastly, our technique works more on reducing coherence traffic of the entire program, rather than identifying individual memory accesses or blocks.

4.6 Conclusion

This chapter presented a profile-guided technique for partitioning memory access operations across distributed data caches to help exploit fine-grain parallelism. The profile gathers statistics on memory operation affinities and working set estimates, and uses the information to create a program-level data access graph. This graph is partitioned to simultaneously cut as few high affinity edges as possible while balancing the working set per cache. The output is a mapping of memory operations to processing elements. By partitioning the memory accesses across distributed data caches, our technique helps to map the problem to similar work in multicluster computation partitioning. The technique can work in conjunction with any standard operation partitioner, and helps to enhance their ability to extract parallelism by proactively accounting for the data memory subsystem. By first making decisions globally on the memory operations, the placement of data became the first-order term in the definition of a good program partitioning. This technique for automated fine-grain parallelization can be used as a complementary method to current coarse-grain techniques, and able to increase the performance of applications on multicore processors. Overall, the profile-guided technique was able to improve the average memory stall cycle time of a standard operation partitioner by 51% and average speedup by 30% for a 2-PE processor.

CHAPTER 5

Region-based Hierarchical Operation Partitioning for Multicluster Processors

5.1 Introduction

The central challenge with clustered architectures is compilation support. The compiler must effectively partition operations across the resources available on each cluster to maximize ILP. However, this goal must be achieved while carefully considering the implications of inter-cluster communication. Communication of values between clusters is both slow and bandwidth-limited. Thus, operations must be partitioned to ensure that ILP is not constrained by frequent inter-cluster communication. A common rule of thumb is that breaking a processor into two identical clusters reduces program performance by around 20%. Furthermore, a four cluster processor loses around 30% performance over the equivalent single cluster processor [18]. Generally, these numbers get worse when the clusters are not identical. While clustering makes sense from an architectural perspective, a large amount of performance is left

on the table with this choice.

This chapter focuses on developing an effective operation partitioning algorithm for a traditional multicluster design. Traditionally, multicluster processors focus on the register file bottleneck by using small decentralized register files. Most designs assume a shared, centralized data memory. For a small number of clusters, using a shared data memory can be a fair assumption. Thus, we first focus our efforts on developing an intelligent compiler partitioning algorithm for a multicluster processor with decentralized register files and a shared memory.

Examining common operation partitioning algorithms in more depth reveals two recurring problems. First, partitioning algorithms are modeled closely after operation scheduling. They make local, greedy decisions to optimize the placement of an operation based on the placement of its neighbors. This strategy makes sense as clustering and scheduling are heavily intertwined. However, locally optimal decisions may actually be poor decisions when the global picture is considered. The second problem is that clustering algorithms are notoriously slow due to the detailed modeling of processor resource constraints. Resource models similar to (or often identical to) those used during operation scheduling are repeatedly evaluated for each candidate operation placement. The final code schedule is indeed very sensitive to the partition chosen, so this seems like the proper strategy. However, detailed modeling of a particular placement can be counterproductive when it limits the number of choices that can be considered. Furthermore, as processors have more resources and their resource usage patterns become more complex, detailed modeling of each placement choice may become infeasible for production compilers.

We use an approach opposite to this scheduler-centric methodology. Operation partitioning is performed at a global scope with the view of all operations in a region (a group of closely related basic blocks is referred to as a region [31]). We adapt two powerful techniques that are commonly used in VLSI design: multilevel graph partitioning and slack distribution. Multilevel graph partitioning divides the dataflow graph into multiple parts in a hierarchical manner. Operations are iteratively partitioned from a coarse level of groups of related operations down to a fine-grained level of individual operations. Slack distribution identifies available scheduling slack within a region and allocates it to specific dataflow edges. In this manner, the cost of cutting specific dataflow edges to create a partition is determined.

Graph partitioning also requires a processor resource model to determine the quality of a partition. Again, we take a non-scheduler-centric approach. We employ a simple estimation strategy that is similar to the RESMII (resource minimum initiation interval) calculation used with modulo scheduling [58]. However, we focus on scalar scheduling as opposed to software pipelining. Resource usage estimates are computed prior to partitioning and used to estimate the resource load for each candidate partition. While this method suffers inaccuracies, it is both more efficient and accurate enough to provide a suitable guide to the operation partitioning algorithm. Our proposed approach is referred to as region-based hierarchical operation partitioning, or RHOP.

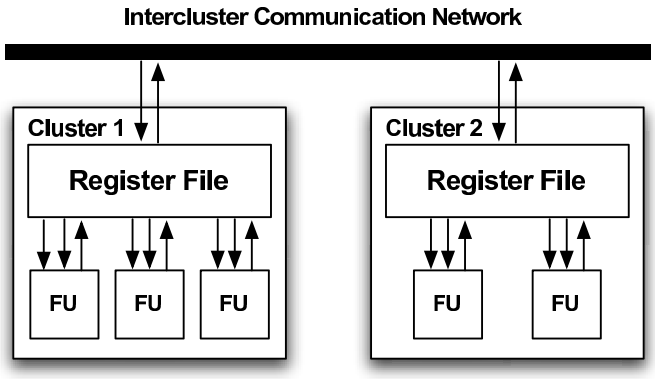

Figure 5.1: A heterogeneous two-cluster machine.

5.2 Overview of Clustering

This section introduces the clustered architectural model that is assumed for this chapter and the basic process of partitioning a dataflow graph (DFG) for this architecture. Next, we present a high-level classification of the common approaches for clustering and break them down by four categories: phase ordering, scope, desirability metric, and operation grouping. Last, we conclude with a discussion of the limitations of scheduler-centric approaches to motivate the work in this paper.

5.2.1 Basics

The architectural model assumed in this chapter is as in Figure 5.1. Each cluster consists of a tightly connected set of register files (RFs) and function units (FUs). FUs in a cluster may only address those registers within the same cluster. Transfers of values between clusters are accomplished through explicit move operations that go through an interconnection network. The interconnection network is assumed to have a uniform connection to all clusters with a fixed bandwidth. Though this assumption

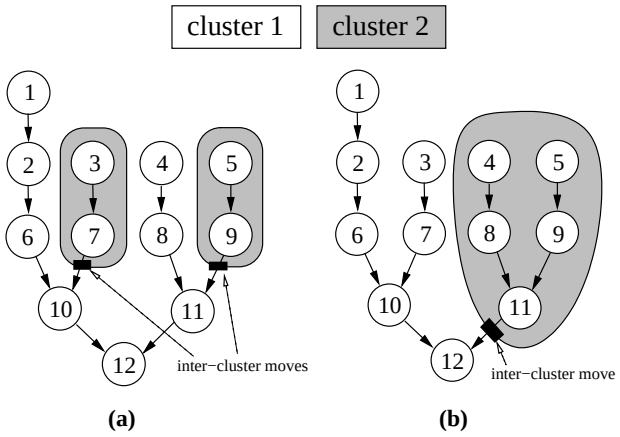

Figure 5.2: Example dataflow graph with (a) locally greedy and (b) region-aware cluster assignment.

is not necessary, it simplifies the compiler algorithms. Clusters in a machine 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 5.1 is heterogeneous and has three FUs and one RF in cluster 1, and two FUs and a RF in cluster 2.

In the presence of a decentralized architecture, the goal for the compiler is to take a graph of the operations and partition them up amongst the resources. Figure 5.2 shows two possible clusterings of an example DFG. For simplicity, the machine that will be used for the example is a homogeneous two-cluster machine, with one RF and one FU per cluster. Each FU is capable of executing any operation. The latencies of all operations are assumed to be one cycle. Furthermore, the interconnection network is capable of sustaining one inter-cluster move per cycle.

For the example in Figure 5.2, the clustering algorithm must partition the operations into two sets with each set executing on a particular cluster. Any time an

edge is broken, an inter-cluster move is required to copy the data from the producing cluster to the consuming cluster. The critical path through this example graph is 5 cycles going through operations 1, 2, 6, 10, and 12. Cutting edges along the critical path increases the critical path length; thus, most clustering algorithms avoid such cuts.

5.2.2 Approaches to Clustering

A large number of algorithms to perform clustering have been proposed by the research community. To better understand the operation and relative strengths of these algorithms, it is useful to understand the major characteristics that differentiate them. We have identified four primary characteristics of clustering algorithms: phase ordering, scope, desirability metric, and grouping.

Phase Ordering. In the compilation process, cluster assignment can take place as a separate phase before scheduling, where the scheduler is constrained by the decisions made in the clustering phase. This isolates the clustering problem from the scheduling problem. Alternatively, clustering can be integrated with the scheduling phase; in this case, more information is available for making decisions, but the complexity of the problem increases, limiting the number of options that can be considered during the process. As a compromise, clustering and scheduling can be done iteratively, such that decisions made by one phase are used to guide the decisions made in the other, until a suitable result is obtained.

Scope. The scope of the clustering algorithm can be local, region-based, or global. A local algorithm generally examines one operation at a time and decides which cluster it should be assigned to based on its immediate neighbors. As with scheduling, operation priorities may guide the order in which operations are considered. A region-based algorithm, on the other hand, considers all of the operations within a region such as a basic block or set of basic blocks at once. Finally, a global algorithm uses knowledge of the entire program or function to make intelligent decisions. The complexity of the algorithm increases with the scope, but better decisions can be made with higher level knowledge.

Desirability Metric. The cluster assignment algorithm can use one of several ways to measure the quality of a candidate partition. It can perform an actual scheduling of the code, which gives the most accurate measure since the performance is then known. It can generate a pseudo-schedule using an approximate machine model to provide a reasonable estimate of the actual schedule. It can use quantitative resource usage estimates to project the load a set of operations places on a cluster. Finally, it can use a simple count of how many operations are on each cluster and how many moves are required to get an idea of the desirability of a partition.

Grouping. Another property of clustering algorithms is whether they employ a hierarchical or flat partitioning scheme. In the case of region-based or global clustering, a hierarchical approach means that decisions are made on multiple levels, with information available in a finer-grained view of the operation graph being used to refine previous decisions made from a coarser view of the graph.

5.2.3 Pitfalls of Scheduler-Centric Approaches

Scheduler-centric approaches to clustering employ a natural extension of the scheduling process to perform cluster assignment. This does not mean clustering is done during scheduling. In fact, any phase ordering is possible. The two distinguishing characteristics of scheduler-centric approaches are local scope and flat grouping. These are the primary characteristics of operation scheduling where operations are greedily placed into the schedule one by one considering the placement of those operations with higher priority that have already been scheduled. The desirability metric for scheduler-centric approaches is most often through the use of an actual schedule. But again, this is not a requirement.

The most well known scheduler-centric clustering algorithm is Bottom-Up Greedy, or BUG [16]. BUG occurs before scheduling; while other algorithms [47, 54, 56], are similarly scheduler-centric though they take place during or interleaved with scheduling. BUG proceeds by recursing depth-first along the DFG, critical paths first. It assigns each operation to a cluster based on estimates of when the operation and its predecessors can complete earliest. These estimates are based on resource usage information from the scheduler, and BUG queries this information twice whenever it considers each operation on each cluster—once before and once after its predecessors have been bound.

This works well for simple graphs, but when the graph becomes more complex such that locally good decisions may have negative effects on future decisions, the algorithm can be fooled into making a bad partition.

Figure 5.2(a) shows a likely operation partitioning generated by BUG or other similar local, scheduler-centric algorithms. The critical path (1, 2, 6, 10, 12) is considered first, and nodes 1, 2, and 6 are placed together on one cluster. Nodes 3 and 7 are placed on the other cluster, since this allows node 10 to begin and complete executing soonest. However, the right subtree is now constrained by the decisions that were locally optimal for the left subtree. As a result, in our example machine which executes one operation per cycle per cluster, this code would take 8 cycles to complete.

The optimal partitioning requires only 7 cycles and is shown in Figure 5.2(b). Operations 4, 5, 8, 9, and 11 should be placed on one cluster, with the remaining operations on the other cluster, as the shading in the figure indicates. With this partition, one inter-cluster move is needed along the edge from 11 to 12. This partitioning minimizes the resultant schedule length by balancing the workload effectively and introducing only one move operation, which is not on the critical path.

Another limitation of BUG is that it keeps track of which resources are busy as it proceeds, and at every step of the algorithm it performs checks to see if a cluster is free at a certain time to perform a certain operation. Therefore, the number of queries to the resource information grows with the number of clusters in the machine and with the number of nodes in the graph.

In order to avoid the potential pitfalls of local decision-making and the compiler overhead of using detailed scheduling information for resources, our approach is to view the graph more globally and to use estimates for determining resource load balance.

5.3 Region-based Hierarchical Operation Partitioning

Our region-based hierarchical operation partitioning algorithm consists of two distinct phases: weight calculation and partitioning. Each operation is represented by a node in a DFG, and node weights are created to represent the resource utilization of the operation. The edges connecting the nodes are given edge weights, which represent the cost on the schedule length for adding an inter-cluster move between those operations. Both node and edge weights are used to guide the partitioning phase. The node weights are used to balance the workload among the clusters, while the edge weights are used to minimize the communication required between them. The partitioning phase consists of a coarsening process, where highly related operations are combined together and placed into clusters, and a refinement process, which improves the initial partitioning. The refinement process uses the calculated weights to consider moving operations between clusters, then iteratively improves the partition by weighing the benefits of improving load balance and reducing inter-cluster communication.

The rest of this section includes a more detailed explanation of the clustering process. The example code and DFG shown in Figure 5.3 will be used throughout this section to demonstrate the process.

5.3.1 Problem Definition

This section provides a formal definition of the graphs and objectives being addressed within this partitioner. For simplification purposes, clusters are assumed to be identical with full interconnect between them.

1. **Data Flow Graph:**

   A graph $\mathbf{G} = (V, E, l, w)$ such that $\mathbf{V}$ is the set of operations in a scheduling region, and $\mathbf{E}$ are the edges, indicating data exchanged between the operations in a producer/consumer relationship. Vertices, $v \in \mathbf{V}$, have a latency of $l(v)$ and edges, $e \in \mathbf{E}$,have a weight of $w(e)$.

2. **Cluster:**

   A cluster $\mathbf{C} = (R, F_1...F_n)$ such that $\mathbf{R}$ is a single register file connected to n functional units $F_1..F_n$.

3. **Machine:**

   A machine $\mathbf{M} = \{C_1...C_k\}$, such that $\mathbf{C_n}$ is one of k disjoint clusters in the processor.

4. **Partition Assignment:**

   A function of the form $\mathbf{P} : \mathbf{V} \rightarrow \{\mathbf{C_1}...\mathbf{C_k}\}$, wherein all the vertices of $\mathbf{V}$ are mapped to a disjoint cluster $\mathbf{C_n}$. Thus, this function maps each of the vertices to one of $\mathbf{k}$ disjoint clusters.

5. **Cut:**

   An edge, $e = (u, v)$ is a cut edge e_{cut} such that, with respect to the partition assignment $\mathbf{P}$, $P(u) \neq P(v)$. Cut edges have a weight $w(e) > 0$, all other edges have $w(e) = 0$.

6. **Intercluster Move:**

   A new operation, v_{icm}, which is inserted into $\mathbf{G}$ between the source and destination operations of e. Each v_{icm} has a given latency $\mathbf{l(v_{icm})}$.

During partition assignment, a valid schedule for a given partitioning must be determined. At that point, a cycle in the schedule is assigned for each operation. Thus, for a given partition $\mathbf{P}$, an instruction schedule is represented by two interrelated mappings:

7. **Instruction Schedule:**

   A mapping of the form $\mathbf{S} : \mathbf{V} \rightarrow \{\mathbf{R_k}, \mathbf{F_n}\}$, such that R_k and F_n are part of a single cluster C_k.

8. **Scheduled Cycle:**

   A mapping of the form $\mathbf{C} : \mathbf{V} \rightarrow \mathbf{N}$, such that each vertex in G is assigned a cycle number in which to execute.

For each of the operations $v \in V$, the dependences between the operations must hold for their scheduled cycle $C(v)$.

9. **Schedule Constraint:**

   Given an operation $\mathbf{v} \in \mathbf{V}$ of the graph $\mathbf{G}$, with the incoming edges:

   $$e_1 = (u_1, v), ..., e_n = (u_n, v), \text{ then}$$

   $$C(v) \geq \max_{i=1}^{n}(C(u_i) + w(e_i)).$$

10. **Schedule Length:**

    The schedule length, $L(S, G)$ is the maximum scheduled cycle of any given vertex in the graph defined as:

    $$L(S, G) = \max_{v \in V}(C(v) + l(v))$$

11. **RHOP Partitioning Problem:** Given a DFG $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, find a partition assignment $P : \mathbf{V} \rightarrow \{\mathbf{C_1}...\mathbf{C_k}\}$ that maps the vertices of $\mathbf{G}$ onto one of the $\mathbf{k}$ disjoint clusters such as the schedule length $\mathbf{L}(\mathbf{S}, \mathbf{G})$ is minimized.

5.3.2 Weight Calculation Phase

Node Weights. The node weights in our graph enable the algorithm to calculate an estimate of how many cycles it will take to execute a set of operations on a cluster, ignoring dependencies, when it is determining the quality of a partition under consideration. Thus, the weights reflect the quantity of resources an operation uses in the machine.

Resources can be characterized as being used by an individual operation, such as FUs, or shared between operations, such as buses or RF ports. In general, resource usage in a machine forms a spectrum between these two extremes. We use the two

endpoints to compute an individual node weight and a shared node weight for each operation. The worst case of these provides an approximation of the operation's resource usage.

Individual node weight, or op_wgt_c, is calculated per node for each cluster c, and is a measure of the resources used by this particular node. Note that, in the case of heterogeneous clusters, the weight of a node is dependent on which cluster it is being considered on. For example, an ADD operation on a cluster with one adder carries more weight than an ADD operation on a cluster with two adders, because in the second case it only uses up half of the available resources.

To determine the weight of a node on a cluster, the number of times that the resources available on that cluster will support the execution of that operation in a single cycle is counted. The weight is the inverse of this number:

$$op\_wgt_c = \frac{1}{\#ops\ supported\ on\ c\ in\ 1\ cycle}$$

Since the example machine executes one operation per cycle, the individual weight of all of the nodes in the graph is 1.0 for both clusters.

To account for shared resources, a shared node weight value is calculated per cluster for the region being clustered. This shared node weight on a cluster, $shared\_wgt_c$, is determined by placing all of the operations in a region on cluster c, and dividing the resulting resource-limited minimum schedule length by the number of operations. The minimum schedule length is similar to RESMII used in modulo scheduling. Since this is done only to determine resource availability and ignores data dependencies, no

actual scheduling is done and the calculation is fast.

$$shared\_wgt_c = \frac{resource\ limited\ sched\ length\ on\ c}{\#ops}$$

In the example, placing the 14 operations on either cluster reveals that a minimum of 14 cycles is required. Thus the shared weight is 1.0 on each cluster. Due to the simplicity of the machine, these numbers are somewhat trivial. Given a real machine with more and varied resources available, as was used in our experiments, the node weights become more interesting.

Edge Weights. The weight of an edge is a measure of its criticalness. If an edge is critical, then placing the nodes on either end of the edge on different clusters and inserting the required move will impact the schedule length. Therefore, edges on the critical path have a higher weight than other edges, and the graph partitioning algorithm attempts to minimize the sum of the weights of the edges that are cut by the partition.

Once critical edges are assigned a high weight, the remaining edges can be assigned a low weight. However, this can be dangerous as there may be a non-critical path that, once a few moves are inserted, becomes critical. Therefore, a more intelligent system for assigning edge weights to non-critical edges is beneficial.

The concept of *slack* is a measure of how critical an edge is. An edge on a path where nodes can be delayed without affecting the overall minimum schedule length has more slack, while a critical edge has no slack. We use a definition of slack similar to that of *global slack* in [21], though our measurement is per-edge rather than per-

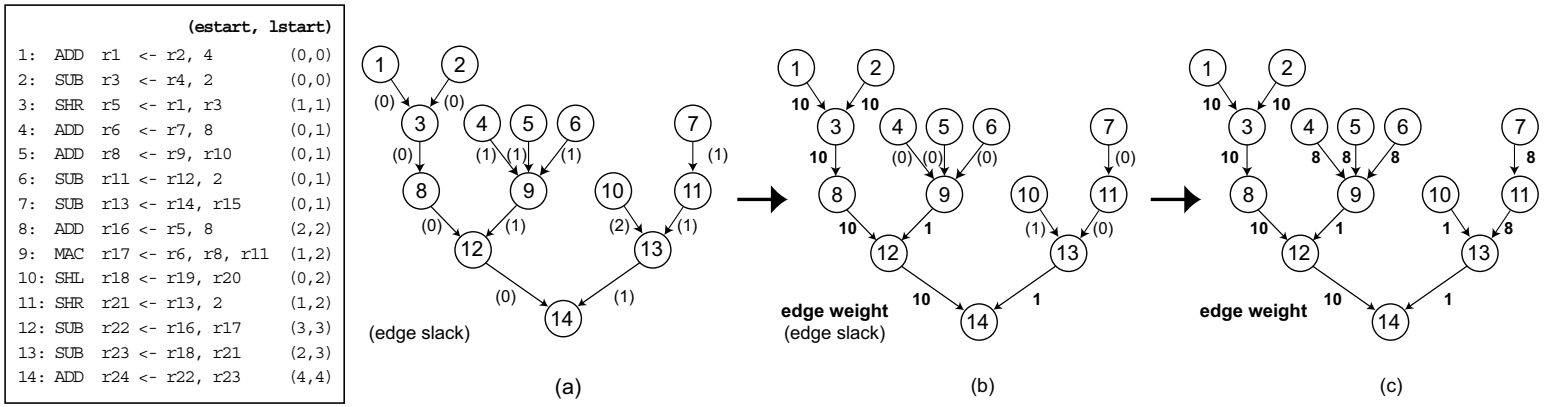

Figure 5.3: The example code and corresponding DFG with slack distribution defining the edge weights.

node.

The slack of a directed flow edge from *src* to *dest* is defined as:

$$slack_{edge} = lstart_{dest} - lat_{edge} - estart_{src}$$

Here, *estart* refers to the earliest cycle an operation can begin executing (i.e. its inputs are available), and *lstart* refers to the latest cycle it can begin executing without delaying the exit operation(s) of the region. The latency of the edge, lat_{edge}, is defined to be the latency of the *src* operation.

Thus, in the example DFG of Figure 5.3(a) with *estart* and *lstart* as shown next to the assembly code, the edges on the critical path 1–3–8–12–14 have zero slack; the edge from node 10 to 13 has a slack of $3 - 1 - 0 = 2$; and the remaining edges have a slack of 1.

A method of first-come first-serve slack distribution is used to account for paths that have some slack in them by taking up slack (by increasing latency) starting from edges close to the critical path. This increased latency may lower the *lstart* of operations higher on the path, thereby decreasing the slack on their incoming edges. The process continues for the next edge on the path until all of the slack has been allocated. The edge weights are assigned depending on whether or not slack was

allocated to the edge, based on the following numbers:

$$edge\_wgt = \begin{cases} 10 & \text{if critical} \\ 8 & \text{if no slack after distribution} \\ 1 & \text{if slack allocated} \end{cases}$$

These numbers were chosen because cutting a critical edge will increase the schedule time so it is weighted high; cutting an edge that has slack allocated to it is essentially free, so it is weighted low. Cutting a non-critical edge that has no slack remaining after distribution is not guaranteed to increase the schedule, but it is likely especially if the "free" edges are cut; therefore, it is given a high weight but not as high as that of a critical edge.

Using this slack distribution algorithm, edges closer to the critical path are more likely to be cut. This accomplishes the goal of offloading as much work as possible from the critical path. It also discourages cutting a single non-critical path too many times such that it becomes critical.

As shown in Figure 5.3(b), the edges which initially had zero slack (indicating that they are critical) are assigned a weight of 10. Now, the non-critical edge from node 13 to 14 can have a unit of slack allocated to it, giving it a weight of 1. This lowers the *lstart* of node 13 by one cycle, with the result that the slacks on the edges coming into node 13 are decreased. Similarly, the edge from node 9 to 12 receives a weight of 1, and the slacks on the edges coming into node 9 are decreased. Figure 5.3(c) shows the final edge weights for this DFG, with the edges that had zero slack remaining

after step 5.3(b) receiving a weight of 8, and the edge from node 10 to 13 receiving a weight of 1.

5.3.3 Partitioning Phase

The partitioning phase of RHOP employs a multilevel graph partitioning algorithm to cluster the DFG of a region into distinct groups. Multilevel graph partitioning, known for its efficiency and good results, is available in many software packages such as Chaco [32] and Metis [38].

A multilevel algorithm coarsens highly related nodes together and places them into partitions. The nodes within the graph are continually coarsened by grouping pairs of nodes together. At each level of coarsening, a snapshot of the currently coarsened nodes is taken. When the number of coarse nodes reaches the number of desired partitions, coarsening stops. The coarse nodes are then assigned to different clusters, and the uncoarsening process begins. During uncoarsening, the algorithm backtracks across the earlier snapshots of coarse nodes, considering moving operations at each stage. A refinement algorithm is used to decide the benefits of moving a node from one cluster to another in order to improve the partition.

5.3.3.1 Coarsening

Coarsening takes a DFG representing the region to be clustered and produces an initial partition for the graph. Producing a good initial partition has been shown to have a large impact on how well the algorithm produces results [32]. The coarsening algorithm uses edge weights determined earlier during the weight calculation phase to

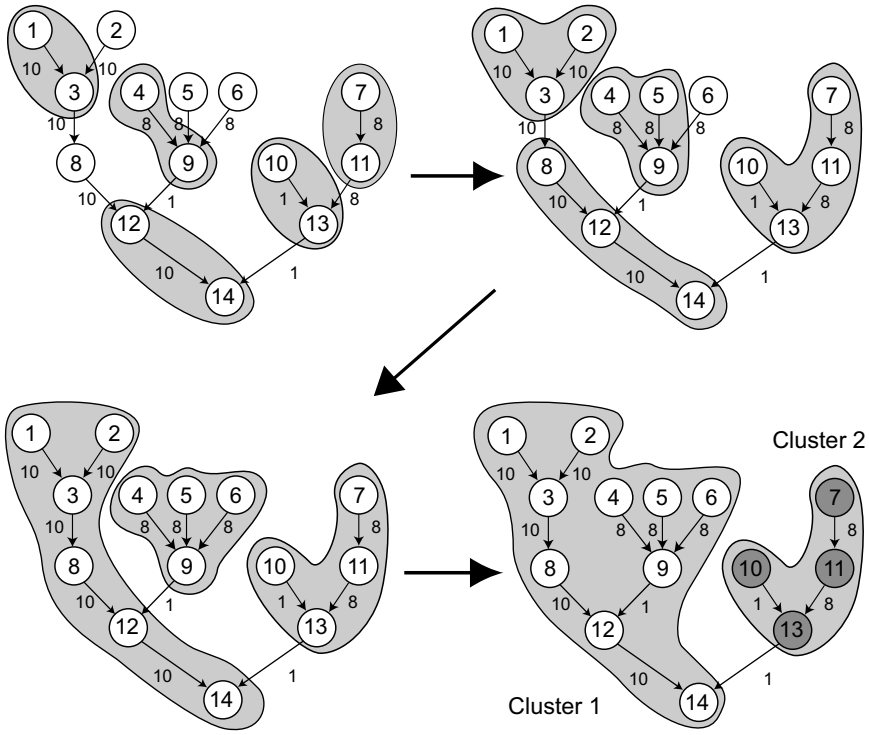

Figure 5.4: The coarsening process to group together highly related operations and create the initial cluster assignment.

intelligently group operations together. Operations separated by a high weight edge are thus first targeted for coarsening.

Each stage of the coarsening process groups together operations into pairs based on the weights of their edges. All operations are sorted based on the highest weight on any of its edges and considered for coarsening in that order. Operations on the critical path will then most likely be paired together. In order to try and coarsen as many nodes as possible at each stage, operations are coarsened from the outside of the DFG toward the center; thus, operations with a single neighbor have higher priority for coarsening. Ties in preferences for coarsening are broken arbitrarily.

Figure 5.4 shows how coarsening progresses through the running example. At the first stage, the first priority is to coarsen together critical edge paths, then the lower

weight edges are considered. Each stage of the process only pairs up a single operation once, and every operation that has an available neighbor to coarsen with will be paired up. Operations that cannot be paired up, either because they have no neighbors or all of their neighbors have already coarsened with other operations, will be left as is for the current coarsening stage. For example, in the first coarsening stage of Figure 5.4, operation 8 no longer has any uncoarsened neighbors, so is not coarsened for this particular stage. When no more operations in a stage can be coarsened, the entire coarsening process is repeated with the resulting coarse nodes.

The coarsening phase ends when the number of coarse nodes is equal to the number of desired clusters for the machine. The coarse nodes are then divided up between the clusters to form the initial partition. For the running example, the final partition ends with operations 1, 2, 3, 4, 5, 6, 8, 9, 12 and 14 on cluster 1; with the rest on cluster 2.

5.3.3.2 Refinement

The refinement process traverses back through the coarsening stages, making improvements to the initial partition. At each uncoarsening stage, the coarsened nodes available at that point are considered for movement to another cluster. In order to properly improve the partition of the operations in the graph, the algorithm must have metrics for deciding which cluster to move from, the desirability of the current partition, and the benefits of an individual move. The refinement process uses the following to judge each of these:

- **Cluster Weight:** The node weights for each operation are used to generate an estimate for the load per cluster; the cluster with the highest weight is denoted the imbalanced cluster.

- **System Load:** Similar to the cluster weight, the system load uses the node weights of all the operations, but estimates the load across all clusters, generating a metric for the current cluster assignment desirability.

- **Gain:** Once the imbalanced cluster has been targeted, the gain of moving each operation to the other clusters is calculated using the change in system load and the change in edge cuts.

Since the process backtracks through the coarsening stages, highly related operations are grouped together at each stage, and the algorithm can then account for a group of operations preferring to move together. This helps to alleviate situations where moving one operation to another cluster is not beneficial, but moving a group of related operations together will show improvements.

The refinement algorithm is a slightly modified Kernighan-Lin partitioner [39], which is known to be a good algorithm for partitioning graphs. In addition, the technique incorporates some of the partitioning improvements implemented by Fidducia-Mattheyses [20]. Traditionally, Kernighan-Lin tries to match pairs of operations from different partitions to swap. Each swap incurs some cost upon the system, and swaps are continually made until the overall cost gain is negative. This allows individual negative moves, which may in fact allow future positive moves to occur. By allowing individual negative moves, the algorithm avoids falling into local minima.

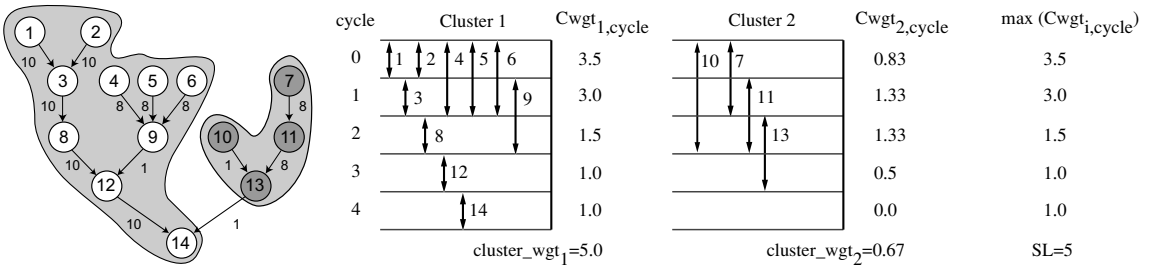

Figure 5.5: The initial partition after coarsening and the cluster weights.

A modified version of Kernighan-Lin is used which considers node and edge weights to determine the gain of moving an operation to another cluster. Unlike Kernighan-Lin, which weighs the benefits of swapping nodes between partitions, RHOP considers explicitly moving each operation within the imbalanced cluster, rather than swapping. Like Kernighan-Lin, RHOP allow moves with negative gain as long as the overall gain for the current refinement step is positive.

Cluster weight. To determine the cluster which is most imbalanced, cluster weights, the metric for the load per cluster, is calculated. In order to calculate the weight of a particular cluster, a weight for each execution cycle of the region is computed. To estimate the weight of each operation at each cycle, we use the scheduling range, which is the *estart* of the operation to its *lstart*. The operation must be placed within this range in order to achieve minimum schedule length.

The two important factors in regards to the load of operations on a cluster are: the individual resource constraints for the operations at each cycle, and the total node weight which is the constraint on the shared resources of a given cluster. The individual resource constraint is the load put on any one specific resource. The shared

resource weight is the load put on all the resources within the cluster as a whole. Since these individual resource and shared resource weights are competing with one another, the overall cluster weight is the *max* between them.

To compute the individual resource constraints, each operation is placed in a single *op group*, which groups similar operations by their resource usage. For each operation in an *op group*, its total impact to a particular cycle is the node weight for the operation, calculated earlier, divided by the slack+1 of the operation. This value, the individual weight, $Iwgt_{c,\tau}$, for cluster c at cycle τ gives a general approximation of the impact of those operations which use a similar resource, currently placed in that cycle on the cluster load.

The total node weight, $Twgt_{c,\tau}$, is then calculated as the total number of operations currently placed within cluster c at cycle τ divided by the average slack for all the operations. This is then multiplied with the shared resource weight from the weight calculation phase to give an approximation of the constraints on the shared resources. This gives an estimate as to how well the operations share the resources at a cycle. The total node weight is effective in situations where there is a lot of parallelism and the assumption of operations finishing within the scheduling range breaks down. Thus, the desired partition focuses more on spreading the work out evenly among the clusters.

The cycle weight $Cwgt_{c,\tau}$ in cluster c at cycle τ is therefore determined by:

$$Iwgt_{c,\tau} = \max_{o \in opgroups} \sum_{op \in o \ at \ \tau} \frac{op\_wgt_c}{op_{slack} + 1}$$

$$Twgt_{c,\tau} = \frac{\#ops\ in\ c\ at\ \tau}{slack_{ave} + 1} * shared\_wgt_c$$

$$Cwgt_{c,\tau} = \max(Iwgt_{c,\tau}, Twgt_{c,\tau}) \tag{5.1}$$

For example, at the end of the coarsening process, the graph reaches a partition as shown in Figure 5.5 with its corresponding cycle-by-cycle representation of all operation scheduling ranges (*estart* to *lstart*). For the simplicity of the example, we will not consider the explicitly consider total node weight, as it has no effect in the result. The $Cwgt_{c,\tau}$ of each cluster is shown in the figure. $Cwgt_{1,1}$, the cycle weight of cluster 1 at cycle 0 is calculated as follows: ops 1 and 2 each have a node weight of 1 and slack+1 of 1. Ops 4, 5, and 6 each have node weight of 1 and a slack+1 of 2. Therefore, ops 1 and 2 each contribute 1 to the cycle weight while ops 4, 5, and 6 each contribute 0.5, which forms a cycle weight, $Cwgt_{1,1}$, of 3.5.

The weight of a cluster, $cluster\_wgt_c$, is simply the sum of all cycle weights from 0 until the max estart, minus one. One cycle is subtracted from each cycle weight to evaluate how overloaded each cycle is. Since every cycle can do one cycle's worth of work, any amount greater than one means the cycle has too much work assigned to it. Therefore, our cluster weight equation is:

$$cluster\_wgt_c = (\sum_{t=0}^{max\ estart} Cwgt_{c,t} - 1) \tag{5.2}$$

System Load. While equation 5.2 gives an estimate to the weight of any one cluster, it doesn't give a general estimate for the overall desirability of the current chosen clustering. This is defined by the system load, SL, which gives a cycle by

cycle account for the clustering. At any given cycle, whenever one cluster's weight dominates that of the other cluster, the smaller load is subsumed by the larger. Therefore equation 5.1, which calculated the load every cycle in a cluster, is maxed it across all clusters and summed for the scheduling range. The system load then results in the maximum any cluster is overloaded over all cycles in the scheduling range.

Since inter-cluster moves have a limited bandwidth, the system load is maxed with consideration of the inter-cluster moves required for this cluster. The inter-cluster move bandwidth, $icm\ bw$, is used to determine the overhead, icm_o, of making the inter-cluster moves required in the current partition. This inter-cluster move overhead is mostly used as a safeguard to prevent clusters from forming that contain far too many inter-cluster moves. In general, the partitioner tries to minimize edge cuts, so this simple estimate of total inter-cluster moves by the cluster is all that is necessary. The system load is therefore determined by:

$$icm_o = \frac{\#icm}{icm\ bw} - (max\ estart\ + 1)$$

$$SL = \max((\sum_{\tau=0}^{max\ estart} \max_{i \in cluster} Cwgt_{i,\tau} - 1), icm_o) \tag{5.3}$$

Gain. At each uncoarsening stage, our algorithm calculates the weight of both clusters using equation 5.2. The cluster with the higher weight, which we refer to as the imbalanced cluster, is chosen as the one to begin moving operations from. A metric is then needed for determining the benefits of moving an operation to a different cluster. Each time a node is moved to another cluster, there is a shift in

both the load balance, as a different set of operations are now on each cluster, and also the cut edges, as there will now be different edges between clusters requiring inter-cluster moves.

Thus, the load gain, $Lgain$, is defined as the difference in the system load before and after the proposed move is made. The edge gain, $Egain$, is the sum of the edge weights of the edges merged minus the sum of the edge weights of the edges cut. The algorithm counts an increase of one on the load gain as equal importance to cutting a critical edge, as it will mean the cycle is so overloaded with work that schedule length must be increased by one. Therefore, the difference in system load is multiplied by the cost of a critical edge, which is currently specified as 10. The overall gain for a particular move, $Mgain$, is then determined by:

$$Egain = \sum_{i \in merged\ edges} edge\_wgt_i - \sum_{j \in cut\ edges} edge\_wgt_j$$

$$Lgain = SL_{(before)} - SL_{(after)}$$

$$Mgain = Egain + (Lgain * CRITICAL\ EDGE\ COST)$$

Figure 5.6 shows the refinement process and gain calculations on the running example. In 5.6(a), the proposed move is to change the coarsened node containing operations 4, 5, 6, and 9 from cluster 1 to cluster 2. This decreases the system load from 5.0 to 4.5, a $Lgain$ of 0.5. By moving this operation over, no edges are merged, and a weight 1 edge is cut (between operations 9 and 12). Therefore the $Mgain$ for this operation is $-1 + 5 = 4$. No other moves in this uncoarsening step are beneficial,

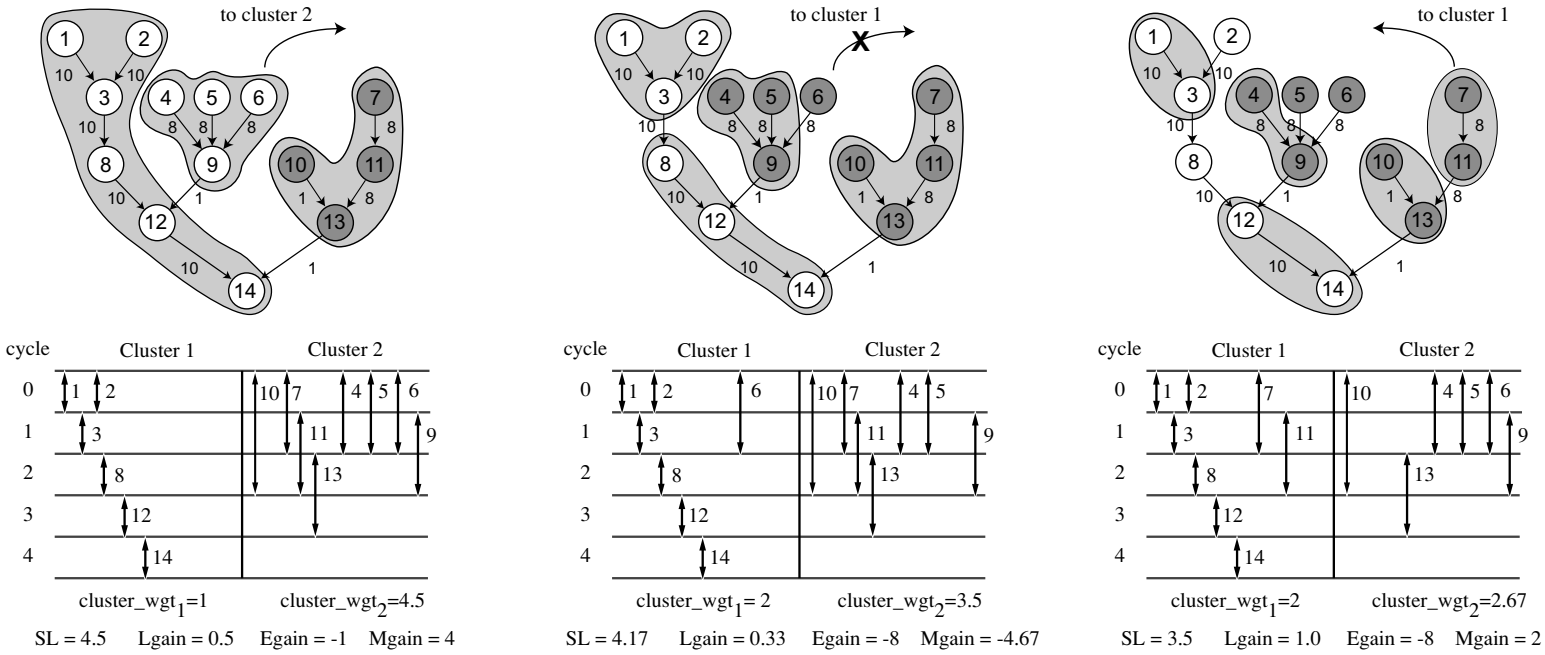

Figure 5.6: The refinement process traveling back through coarsened states. (a) The beneficial move of the coarsened node containing operations 4, 5, 6 and 9 to cluster 2. (b) A situation where no positive moves exist and the move is not made. (c) Moving the coarsened node containing 7 and 11 to cluster 1 is now beneficial.

so the graph is uncoarsened again.

The next uncoarsened state is shown in Figure 5.6(b), where operation 6 has been uncoarsened from 4, 5, and 9, all of which are now in cluster 2 after step 5.6(a). Of interest is that even though moving operation 6 from cluster 1 to cluster 2 provides a positive *Lgain* by dropping the system load from 4.5 to 4.17, This is not enough to counteract the cost of cutting the weight 8 edge from operation 6 to 9, therefore this move is not made. Since no move in this coarsening state is beneficial, uncoarsening continues.

Next the graph reaches the uncoarsening state in Figure 5.6(c). At this stage of uncoarsening, both the coarsened nodes containing 4 and 9 as well as 7 and 11 decrease the system load from 4.5 to 3.5 if moved to cluster 1, as they both have the same node weights and affect the same cycles. The coarse node with 7 and 11 is chosen for moving, though, because it only cuts one weight 8 edge and thus remains a positive move, while the coarse node 4 and 9 cuts two weight 8 edges and merges one weight 1 edge, making it a negative $Mgain$.

Each uncoarsening stage finishes when it can make no more moves and the same imbalanced cluster is chosen twice in a row. Then, it moves on to the the next uncoarsened stage and the refinement process is repeated. When the uncoarsening process completes its refinement of the original, totally uncoarsened snapshot of the region, one final pass through each of the clusters is run, to ensure that no positive moves out of a cluster were ignored because another cluster was extremely out of balance. In this final phase, each cluster allows only positive moves.

When this final phase completes, the resulting partitions correspond to the desired clusters. The node weights and edge weights ensure that there exists a good load balance between the clusters as well as a minimal cut set for inter-cluster communication. For this example, the final uncoarsening step yields no change from the partition after the move in Figure 5.6(c). Thus, the final partition is operations 1, 2, 3, 7, 8, 11, 12 and 14 on cluster 1, with the remaining operations on cluster 2. Even though there are three cuts in this partition, none are critical and this results in the optimal schedule length of 8 cycles.

5.4 Experimental Evaluation

We implemented the RHOP algorithm using the Trimaran tool set [67], a retargetable compiler for VLIW processors.

5.4.1 Methodology

To gauge the performance of our algorithm, we compared our results to the BUG algorithm. We evaluated the performance of both BUG and RHOP on several DSP kernels and the SPECint2000 benchmark suite. DSP kernels were investigated because of their characteristically high ILP that make them ideal candidates for wide-issue processors. As a result, they provide a true measure of the clustering algorithm's ability to exploit high levels of ILP. The SPECint2000 benchmarks[1] were also used because of their generally low and irregular ILP. These benchmarks provide the chal-

[1] 176.gcc, 186.crafty, and 252.eon were not run due to limitations of the current Trimaran compiler system.

Name	# Clusters	Configuration
2-1111	2 Homogeneous	1I, 1F, 1M, 1B per cluster
2-2111	2 Homogeneous	2I, 1F, 1M, 1B per cluster
4-1111	4 Homogeneous	1I, 1F, 1M, 1B per cluster
4-2111	4 Homogeneous	2I, 1F, 1M, 1B per cluster
4-H	4 Heterogeneous	IF, IM, IB, and IMF clusters

Table 5.1: Our clustered machine configurations.

lenge of exploiting ILP when it is available, but not over-partitioning when ILP is limited.

Five different machine configurations were used to compare our performance with BUG. The details of these machines are shown in Table 5.1. Common to all these machines are 64 registers per cluster, operation latencies similar to those of the Itanium, and perfect caches. Four of the machine configurations have homogeneous clusters (i.e. the resources on each cluster are identical), and the last one is a heterogeneous machine. Each has varying numbers of integer (I), float (F), memory (M) and branch (B) units. The different machine configurations are summarized below:

For each benchmark, the dynamic cycle count was used as the evaluation metric for how well the clustering algorithm was able to partition the code into clusters. After clustering, prepass scheduling, register allocation and postpass scheduling are performed to generate the final assembly code.

5.4.2 Performance Improvement

Table 5.2 shows our improvement over BUG for 13 kernels and the SPECint2000 benchmarks for the five different machine models. For each kernel, we present the

Kernel	2-1111	2-2111	4-1111	4-2111	4-H
adpcm	-2.09	3.25	12.03	8.95	11.98
atmcell	-0.32	3.34	34.86	32.58	14.04
channel	-3.35	-0.73	11.20	20.44	6.50
dct	-0.64	10.53	31.24	28.86	17.31
fir	4.75	15.74	30.90	12.34	11.62
fsed	4.39	6.52	22.87	27.90	10.65
halftone	1.17	4.91	27.99	34.18	-2.12
heat	-6.24	21.50	31.23	33.32	15.26
huffman	-4.84	-3.87	24.65	24.79	19.76
LU	-2.65	-1.23	-1.42	12.44	4.51
lyapunov	1.83	9.43	13.26	6.63	13.41
rls	-1.90	4.50	6.09	30.51	11.42
sobel	-2.04	1.02	20.67	20.92	22.20
Average	-0.92	5.75	20.43	22.60	12.04

SPEC	2-1111	2-2111	4-1111	4-2111	4-H
164.gzip	-2.18	5.21	8.67	6.86	4.12
175.vpr	-5.98	2.42	3.26	6.15	3.41
181.mcf	-1.72	-1.49	3.99	-5.99	-3.44
197.parser	-3.45	-2.76	-1.22	-1.40	-1.62
253.perl	-3.16	0.00	6.25	5.13	1.84
254.gap	-5.79	0.34	-0.76	0.47	-1.08
255.vortex	-2.61	2.08	-5.29	7.59	1.84
256.bzip2	-1.02	-0.29	25.45	21.66	9.66
300.twolf	-2.16	1.13	8.24	3.41	4.04
Average	-3.11	0.73	5.40	4.87	2.28

Table 5.2: Percentage improvement by RHOP on cycle time over the BUG algorithm for several kernels and the SPECint2000 benchmarks on five different machine models.

percentage improvement in dynamic total cycles of RHOP over BUG. Positive results mean RHOP performed better than BUG, while negative results mean BUG performed better.

Overall, our results for a two cluster machine with one resource of each type are rather poor, with an average increase in dynamic cycles of 0.92% on the kernels. As the machine configuration becomes more complex, by adding more resources and additional clusters, results dramatically improve in the quality of our operation clustering. On the kernels, there was an average of 20% improvement on the 4-1111

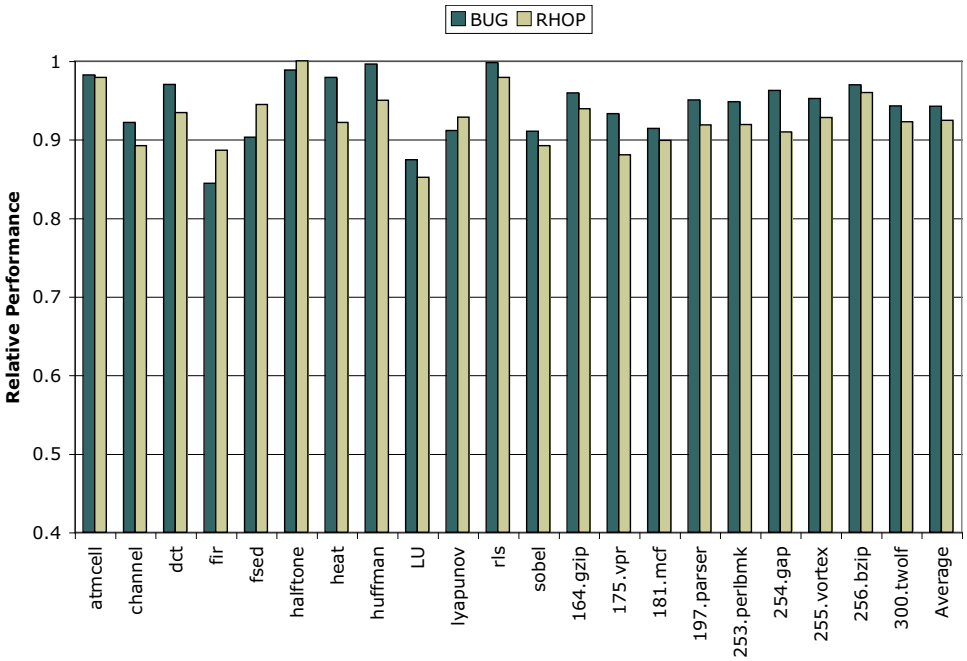

Figure 5.7: Comparison of BUG and RHOP clustering performance degradations on a 2 cluster (2-1111) machine configuration versus a 1-cluster (1-2222) machine with the sum of the resources of the clusters.

machine, and a 23% improvement on the 4-2111 machine. The results for the four-cluster heterogeneous machine fell between the two and four-cluster homogeneous machines. A similar trend is seen on the SPECint2000 programs in Table 5.2 except the improvements are more modest. In general, the SPECint2000 benchmarks have less ILP than the kernels, thus there is less opportunity for distributing work across clusters.

The data in the table shows that local, greedy methods for clustering can perform quite well in constrained, resource limited situations. The poor results from the two-cluster machine occur because of the inaccuracies of our resource model. Estimates, in general, can be wrong, and at times we observe one cluster gets more operations than it should. One major factor is that our resource load estimate ignores edges;

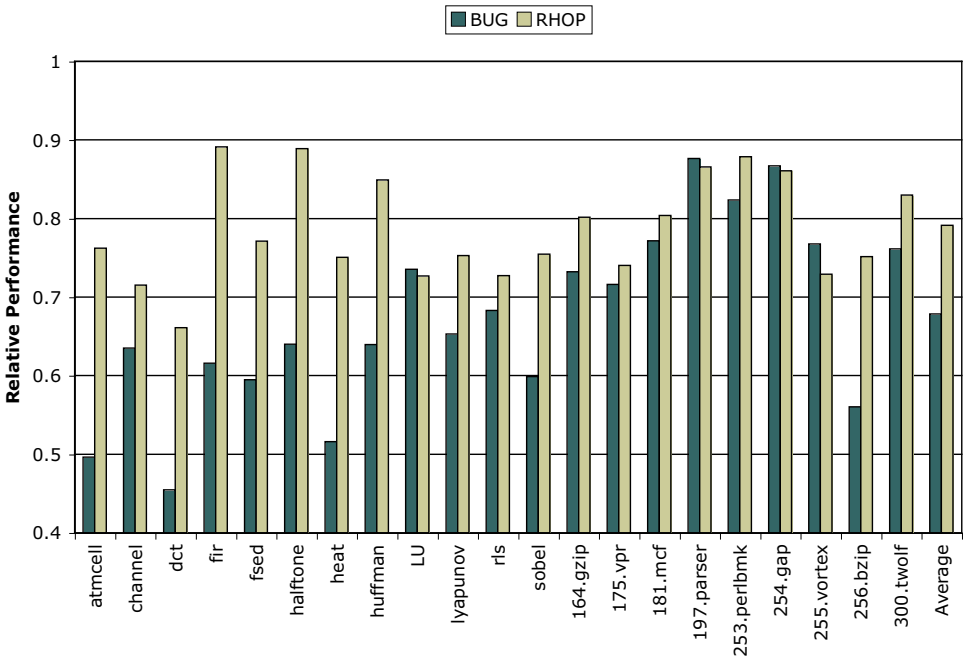

Figure 5.8: Comparison of BUG and RHOP clustering performance degradations on a 4 cluster (4-1111) machine configuration versus a 1-cluster (1-4444) machine with the sum of the resources of the clusters.

thus, it also ignores dependencies between instructions, and assumes reordering is possible where in actuality, it may not be. We then have too many operations being placed into a cluster and forced to execute serially.

On the other hand, for four-cluster machines, the most important factor is carefully spreading out the workload among all the different clusters. In such a situation, the region-level scope used by RHOP becomes much more effective than a local, operation-centric scope. Thus, we are able to achieve a drastic improvement for four clusters.

Figures 5.7 and 5.8 compare the performance of the 2-1111 and 4-1111 machines using BUG and RHOP with a single cluster machine containing the sum of the resources of all the clusters, respectively. The single cluster machine provides an upper-bound of performance. For the two-cluster results, both BUG and RHOP achieve

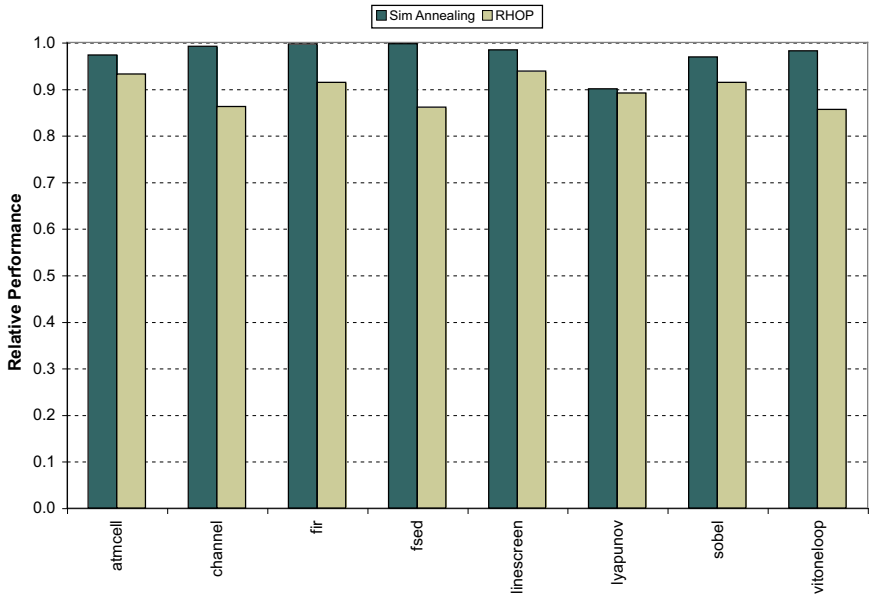

Figure 5.9: Comparison of our RHOP technique with a simulated annealing method.

greater than 92% of the upper-bound. Conversely, for a four-cluster machine, BUG only achieves 68% of the upper-bound. Again this is due to the local, greedy heuristics breaking down for wide machines. RHOP increases performance to 79% of the upper bound. Clearly, there is still room for improvements to the RHOP algorithm.

5.4.3 Comparison to Simulated Annealing

The previous graphs compared our RHOP technique to a single-cluster machine with the sum of the resources. This baseline single-cluster machine is an unachievable upper-bound, since it has all the resources of the clustered version, but does not suffer from the additional intercluster move latency. To find what a closer upper-bound for the optimal partitioning assignment, we developed a technique which anneals the partition assignment and uses the actual scheduler to measure the quality of the partition. Thus, this partitioner could only work on a few smaller benchmarks

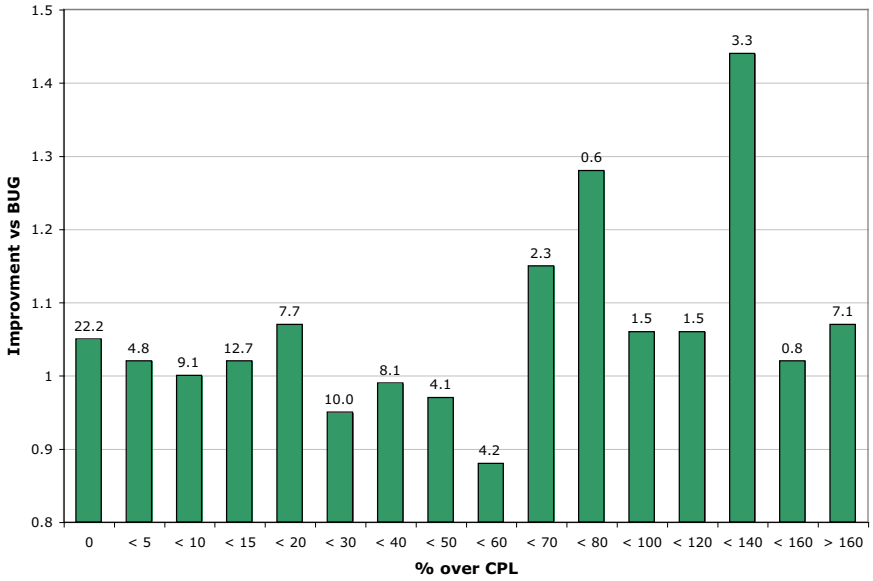

Figure 5.10: Histogram comparing the performance of RHOP and BUG; each category is the achieved schedule length of the region with respect to the critical path length. The numbers on top are the dynamic execution percentage of the category.

because of its significant compiler runtime. The results for this technique are shown in Figure 5.9, where we compare the simulated annealing technique and RHOP against the baseline single-cluster performance for a 2-cluster machine. This graph shows that our partitioner is doing quite well in these benchmarks, and is approximately 93% of the performance of the simulated annealing technique, on average.

5.4.4 Compile-Time Effects

Since our desirability metrics assume that schedules finish within the critical-path length (CPL) number of cycles, we performed a study of RHOP performance as a function of schedule length relative to the CPL. In Figure 5.10, each bar represents the cumulative ratio of RHOP cycles over BUG cycles for all regions (across all benchmarks) in the range. The bars are annotated on top with the percentage of

Kernel	Sched	BUG	RHOP
adpcm	4550	13777 (3.0)	6676 (1.5)
atmcell	31560	109880 (3.5)	33244 (1.1)
channel	12294	32094 (2.6)	14686 (1.2)
dct	16646	47346 (2.8)	17148 (1.0)
fir	9284	26434 (2.8)	10474 (1.1)
fsed	13300	40244 (3.0)	13910 (1.0)
halftone	14109	29519 (2.1)	15475 (1.1)
heat	5159	13113 (2.5)	5667 (1.1)
huffman	22030	54974 (2.5)	25296 (1.1)
LU	1935	4563 (2.4)	3105 (1.6)
lyapunov	16256	43234 (2.7)	17940 (1.1)
rls	25305	84413 (3.3)	26471 (1.0)
sobel	8138	23145 (2.8)	9414 (1.2)
Average		(2.8)	(1.2)
SPEC	**Sched**	**BUG**	**RHOP**
164.gzip	385173	1303443 (3.4)	455795 (1.2)
175.vpr	1356211	4555987 (3.4)	1528947 (1.1)
181.mcf	220845	700958 (3.2)	245269 (1.1)
197.parser	1238238	4045074 (3.3)	1434704 (1.2)
253.perl	2102449	7066202 (3.4)	2862355 (1.4)
254.gap	2046872	6754402 (3.3)	2813026 (1.4)
255.vortex	2133516	7199868 (3.4)	2635402 (1.2)
256.bzip2	489923	1580643 (3.2)	550493 (1.1)
300.twolf	1405475	4861800 (3.5)	1681433 (1.2)
Average		(3.3)	(1.2)

Table 5.3: Number of calls to the resource table. For BUG and RHOP, the ratio of total calls over Scheduling-only calls is given in parentheses.

dynamic cycles that occur within these ranges. For regions very close to the CPL, our algorithm performs modestly well. These regions are critical-path limited, and our system load estimates are quite accurate. For regions much higher than the CPL, where the regions are resource-constrained, our algorithm performs even better. In such regions, the key to a good partition is properly spreading out work across the clusters, and the total node weight heuristic in RHOP intelligently balances the workload. The middle ground, when regions are neither critical-path nor resource constrained, is where RHOP has the most difficulty. Since neither resources nor CPL

dominate, our resource estimates lose substantial accuracy and thus bad clustering decisions can be made.

In addition, the runtime of the two algorithms was evaluated. For a research-oriented compiler like Trimaran, simply evaluating raw compute time is a rather inaccurate way to measure the speed of an algorithm. A more realistic measurement is the number of calls to the resource table, which gives an estimate on how often the algorithm is checking and rechecking its resource model. This is the heart of the scheduler, where most of the time is spent. Thus, minimizing entries into this function is a key metric to improving compiler run-time. The results from this experiment are presented in Table 5.3. Our algorithm shows significant improvement over BUG, taking an average of 1.2 times the runtime of the scheduler alone, versus 3.0 times for BUG. This is a result of the necessity of scheduler-centric algorithms requiring a detailed model of the current resource constraints and repeatedly reevaluating the model for each step of the process.

	When (rel. to sched)			Scope		Desirability Metric				Grouping	
Algorithm	Before	During	Iterative	Local	Region	Sched	Pseudo	Est	Count	Hier	Flat
BUG [16] [52]	√			√		√					√
PCC [14]	√			√	√	√				√	
UAS [56]		√		√		√					√
ABC [41]	√				√				√		√
Eichenberger [54]			√	√		√					√
Leupers [47]			√	√		√					√
Capitanio [6]			√		√				√		√
URACAM [12]		√		√			√				√
GP(A) [1]			√		√	√				√	
GP(B) [2]		√			√		√			√	
B-ITER [42]	√			√				√			√
CARS [37]		√			√			√			√
Convergent [46]		√			√			√			√
RHOP	√				√			√		√	

Table 5.4: A comparison of several different clustering techniques based on four important characteristics: when the clustering occurs in relation to scheduling, the scope of the algorithm, the metric used in order to determine the quality of the partition, and whether operations are considered individually or in groups.

5.5 Related Work

There has been a large body of research conducted in the area of clustering. In Table 5.4, we summarize our general categorization of many of them based on the four characteristics of clustering algorithms presented in Section 2.

The most closely related work to our clustering approach is with algorithms that use graph partitioners, and those that use an estimate-based approach. Capitanio et al. [6] proposed a graph partitioning method to clustering operations, but focused mainly on a Kernighan-Lin like approach to improving partitions. They focus their improvements strictly on a function of the partition cut set, and weighing the benefits of making a cut with the probability that it will increase the schedule length.

Aletà et al. use a similar multilevel graph partitioner, but focus on tightly integrating the clustering algorithm with the instruction scheduling and register allocation [1]. Also studied was clustering via a multilevel partitioner to determine the optimal initiation interval (II) for a modulo scheduled loop using a pseudo-scheduler [2]. Their work focuses on scheduling cyclic code in multicluster domains, while ours is targeted toward acyclic code. We also use substantially different models for computing node and edge weights.

While not a heavily researched area, there has been some work on estimate-based approaches for clustering. Lapinskii et al. [42] base their estimate off three major factors: the data transfer penalty, FU serialization penalty and bus serialization penalty. They use a local approach like BUG to minimize the data transfer penalties. The FU serialization is basically the load of the cluster, which is determined in a cycle by

cycle approach.

Partitioning can also be approached by considering operands instead of operations [34]. Research on partitioning for multiprocessors has many similarities to clustering for multicluster processors. Yang and Gerasoulis [68] proposed a low-complexity method for clustering and scheduling parallel tasks for multiprocessors. Liou and Palis [50] improved upon the complexity of this algorithm.

More generally, graph partitioning is a widely studied area, and many similar problems have been mapped to graphs in the past. The ares of floorplanning and placement have been using graph partitioners to reduce their problem sizes. For example, Li et al. [49] present a technique for efficiently partitioning a graph through a multilevel algorithm. These floorplan partitioning techniques are similar to our problem in that the graph objective function is difficult to calculate simply by examining node weights and edges, and must be heuristically approximated. The problems differ in the basic nature of the optimization problem as well as the problem sizes. Floorplanning usually has much larger graphs, while DFGs of applications are typically smaller and narrow as they are linearized by the scheduler.

5.6 Summary

This chapter introduced a novel technique to cluster operations for multicluster processors with a shared data memory. A slack distribution algorithm is presented, which effectively weights edges based on their preference for being broken across clusters. We introduce a new way to estimate the impact of clustering decisions,

which is used to guide our graph partitioner. Our graph partitioner is able to consider an entire region of code and base its decisions off a view of the code as a whole, rather than what the best clustering is for a single operation.

We compared our results to a popular algorithm, BUG, and results show that for larger number of clusters, our algorithm is able to efficiently produce better partitions. Two-cluster machines saw an average performance decrease of 1.8% across all kernels and benchmarks. As we increased the number of clusters, there was a dramatic increase in the performance of our partitioner. A four-cluster machine provided an average improvement of 14% in our experiments.

CHAPTER 6

Data and Computation Partitioning for Fine-grain Parallelism

6.1 Introduction

In recent years, the processor design industry has shifted away from increasing the performance of monolithic, centralized processor designs. In the past, processor generations could scale performance by increasing clock frequency and designing larger, more complex structures. However, problems with increased power dissipation and thermal issues have become the main design constraints, forcing a change to decentralized multicore designs. Multicore processors lessen the power issues by using multiple simpler cores and tightly integrating them together on a single die. This allows for an increase in throughput capabilities of the processor but does not necessarily increase performance. As Moore's Law continues to increase transistor counts, the semiconductor industry is expected to use the additional transistors to scale the number of cores per chip from the 2 to 8 core processors currently available

on the market to many more in the future.

While the shift to multicore designs has the ability to significantly improve the performance of applications, this performance boost is not free. In order to take advantage of massively parallel cores, the difficult problem of parallelizing an application falls back on the programmer and compiler. Traditional code generation for decentralized multicore processors focus on coarse-grain methods for parallelization. These techniques include new programming models [7, 30, 33, 51] and different ways to exploit thread-level or single-instruction multiple-data (SIMD) data-level parallelism [15, 28, 43, 61]. While these methods are an extremely effective way for programmers to increase overall throughput of the processors, there is still available parallelism which can be extracted by the compiler. In addition, there is also a significant number of single-threaded applications and programs that simply do not exhibit the inherent parallelism for programmers to widely spread their execution across multiple cores. However, at a fine-grain level, the compiler can determine parallel tasks to distribute across the cores and further increase performance.

This chapter focuses on a two-phased cooperative compiler-directed method for program parallelization by exploiting fine-grain instruction-level parallelism (ILP). Current research in interconnection networks has examined multiple ways to increase the speed and bandwidth of communication between cores [57, 64]. Faster communication of scalar values between the cores enables applications to take advantage of parallelization at the operation granularity. While coarse-grain techniques can parallelize large portions of execution, our method can use an additional dimension to further increase performance by creating fine-grain threads to exploit on the multiple

underlying cores.

The challenge for exploiting fine-grain parallelism is: given an application, identify the operations that should execute on each core. This decision must take into account the communication overhead of transferring register values between the cores as well as the layout of data values in the individual caches of each core. Poor decisions could lead to communication across the interconnection network delaying the execution of other operations, cache conflicts evicting data and increasing cache misses, or an increase in cache coherency traffic between the cores, all of which would lead to lower performance. The fine-grain nature of these decisions make it difficult for the programmer to specify. However, the compiler is able to take advantage of analyses of the data-flow and memory access behavior to make better decisions to how to distribute the application.

Extracting fine-grain parallelism is a difficult task, but as the industry moves to faster, tighter interconnection networks between the cores, many similarities can be drawn with multicluster processors in the embedded domain. These processors address the issue of fine-grain parallelism by relying on the compiler to partition operations across the multiple clusters [1, 6, 16, 37]. The main architectural difference between multicluster and the multicore processors of today are that multicluster designs generally have a shared data cache, while multicore systems have coherent distributed data caches per core. This adds another level of complexity for the compiler to be cognizant of data values and how they are brought into each individual cache. However, by phase-ordering the compiler passes to first partition the data, then partition the computation cognizant of the data location, our technique simpli-

fies the problem while still generating high-performance code. This chapter brings together the data partitioning techniques from Chapters 3 and 4 with the computation partitioner described in Chapter 5 and details how they interact to produce a partitioned application for decentralized architectures.

6.2 Fine-grain Parallelism Extraction

Both data and computation partitioning are important compiler phases to extract fine-grain parallelism for decentralized processors. Chapters 3 and 4 explained two possible techniques for partitioning data memory across decentralized scratchpad memories and data caches, respectively. Chapter 5 introduced a region-based technique for partitioning computation operations across multiple PEs. In this section, the combined technique is explained which phase orders data and computation partitioning. The data partitioning is first completed, and passes its decisions down to the computation partitioner, which can factor in the location of data when making its partitioning decisions.

6.2.1 Data-cognizant Computation Partitioning

A phase-ordered partition of data and computation can occur either data-first or computation-first. In this dissertation, we made the partition of the data the first-order term in producing the overall partition. Data is more global in nature and the effects of memory stalls can be much greater. In addition, problem size of data partitioning was small enough to enable the compiler to make global, program-level

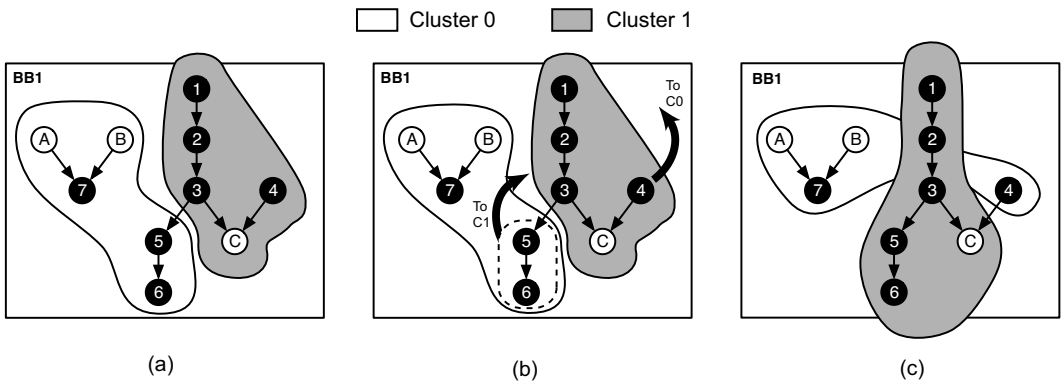

Figure 6.1: Example of computation partitioning where shaded areas are operations in cluster 1: (a) the cluster assignment designed by the first-pass data partitioning. (b) two performance based improvements made by the computation partitioner and (c) the final partition.

decisions. There are far more computation operations, making program-level analysis infeasible.

The process of making the second computation partitioning phase cognizant of the data required the first phase to produce a mapping of data access to cache. Thus, our compiler phase determines a memory access partition and locks, or freezes, the memory operations to a specific PE. The second phase, RHOP, can easily take this prepartition of the data into consideration, by not moving the memory operation and understanding the required interconnect moves needed to transfer the data to any desired PE. This also gives RHOP the ability to add preference for specific computation operations to move closer to their data location.

Figure 6.1 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 a the single block of code, we see the partition of operations breaking the edge between operations 3 and 5, as shown in Figure 6.1(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 6.1(b). The final partitioning is shown in Figure 6.1(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.

6.3 Experiments

To study the overall effects of fine-grain parallelism extraction, we ran our cooperative data and computation partitioning technique on 2 and 4 core machines. Each core can execute one integer, floating point, memory and branch operation per cycle. The cache configuration for these experiments was set at 4kB per core. The reported results are through the Trimaran compiler and simulation infrastructure with the M5 cache simulator. The details of the simulated machines are presented in Table 6.1.

Parameter	Configuration
Number OF PEs	2, 4
Function Units	1 I,F,M,B per PE
PE Comm. B/W	1 total move per cycle
PE Comm. Latency	1 cycle
L1 Cache	2-way associative
L1 Block Size	32 bytes
L1 Cache Sizes	4kB per PE
L1 Hit Latency	1 cycle
L1 Bus Latency	2 cycles
L2 Hit Latency	10 cycles
Main Memory Latency	100 cycles
Coherence Protocol	MOESI

Table 6.1: Details of the simulated multicore machine configuration

This section presents the achievable speedups from increasing the number of cores and utilizing the extra resources with fine-grain threads.

6.3.1 Partitioning for 2 Cores

Figure 6.2 shows the achieved speedup over a single PE machine on a 2-PE processor and our fine-grain parallelization techniques. Thus, 1.0 in the x-axis indicates the performance of a single PE processor, and higher bars indicate better performance. For each benchmark, three bars are shown. The first indicates the performance achieved by a data-incognizant partitioner which purely focuses on the computation operations. This technique generally does the worst, as it suffers from a poor data access distribution and memory stalls. The second bar indicates the performance of our technique, where we proactively distribute data accesses. The final bar is the performance of a unified machine: a single PE processor with twice the resources. Thus, this bar is an indication of the upper bound of our technique, as it

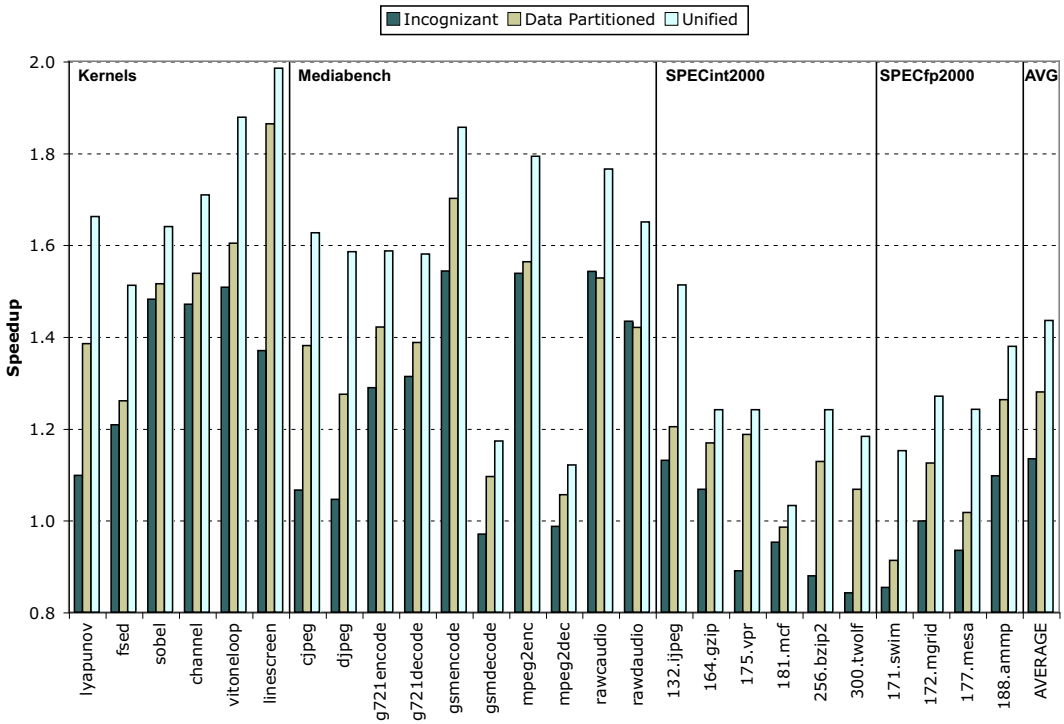

Figure 6.2: Speedup over a 1-PE processor when using 2 PEs and our data partitioning technique.

can support the same amount of parallelism and does not suffer from the intercore communication latencies.

Overall, most benchmarks show a performance improvement. As expected, benchmarks with more parallelism such as the kernels and Mediabench, show more speedup. This is directly related to the amount of parallel work available for our fine-grain technique to exploit. The SPECcpu benchmarks show less speedup, but the unified machine speedup for each benchmark indicates that many of those applications have very little room for improvement. On average, we saw the upper-bound of achievable speedup around 1.43 and our techniques able to extract approximately a 1.3 speedup with our data access partitioning.

It is evident that a data-incognizant partitioning is not a good solution, and the

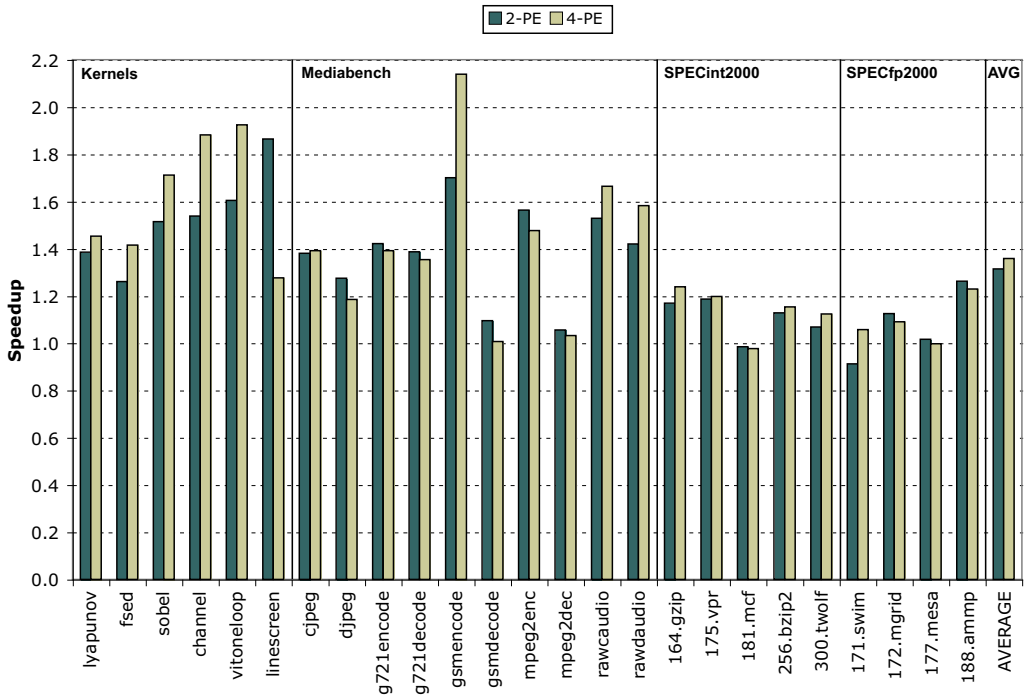

Figure 6.3: Comparison of 2-PE and 4-PE machines for overall speedup

proactive distribution of the data is extremely important in leveraging benefit from the extra resources. In many cases, such as *gsmdecode*, *175.vpr*, and *300.twolf*, a decrease in performance is shown for a data incognizant partition, as the memory stall time takes away any gains from computation parallelization. However, with our proactive data partitioning, all three of these benchmarks show some speedup over the baseline case. Two benchmarks, *181.mcf* and *171.swim*, show slight performance degradation even with our technique. In these applications, the amount of parallelism available was so low that we were not able to extract enough parallel work.

6.3.2 Partitioning for 4 Cores

To study the scalability of our technique, we partitioned each program to 4 PEs to see how much additional performance benefit existed. Figure 6.3 shows that overall speedup was also similar to our 2-PE results. Some benchmarks, such as *gsmencode*, where dramatically improved; however, on average, there wasn't a significant increase in performance even though were four times the number of resources as the baseline case. Much of the achievable performance benefits depends on whether or not the benchmark has enough parallelism to support the wider machine. In addition, the larger number of PEs increases the contention for the communication bus causing more compute cycles to be executed. This shows us that while fine-grain parallelism is useful and can be exploited for performance improvement, it has its limits based on the application.

6.3.3 Conclusion

This chapter presented a methodology for incorporating both data and computation partitioning into a single compiler process for extracting fine-grain threads of execution. With multicore processors becoming more commonplace, and numbers of cores expected to rise, both the compiler and the programmer need to develop new ways to exploit parallelism in their applications. While the major gains must be made at the coarse-grain level, the work presented in this chapter shows how fine-grain examination of applications can be an complementary and effective way to further increase performance. Compiler-directed parallelism extraction first divides

the memory accesses across the cores, cognizant of the affinity relationships between different memory operations. This is followed by a detailed computation partitioning pass which understands the location of each memory access and can take the data transfer latencies into account when creating the fine-grain threads. Overall, we found that many applications have enough inherent parallelism to utilize two cores; however, this did not scale in all applications to four cores. Thus, fine-grain parallelism extraction can be an useful technique to increase performance, but each application has its limit on effectiveness.

CHAPTER 7

Conclusion

As decentralized processors have become ubiquitous in both the general-purpose and embedded-system domains, techniques to exploit underlying application parallelism have become vital to achieve high performance. Decentralized processors partition the processor resources, such as FUs, register files and data memories, into smaller subsets connected together with a communication network. Thus, performance in these processors depends more on throughput and parallel execution than raw clock frequency. This dissertation focuses on compiler processes to analyze and develop efficient fine-grain partitions of both the data and code of a program, allowing for effective parallel execution and increased performance.

7.1 Summary

The partition of the data is the first phase of the code generation system, and is expected to produce a mapping of memory access operations to PEs. This dissertation proposes two methods for providing this division of the memory operations: static and

profile-based techniques. The static method analyzes the application to determine the set of objects accessed by each memory operation. Given this knowledge of statically accessed objects, a partitioning of the objects to memories could be made, which, in turn, led to a mapping of the load and store operations for those objects. A partition of the objects worked well in the embedded design space with scratchpad and static local memories, achieving up to 99% (96% on average) of the performance of a larger, shared memory when using small, decentralized memories.

While this static technique for object partitioning is an efficient compiler process, it did not translate well to the general-purpose domain with caches. Partitioning an entire object to a specific cache in one PE does not effectively utilize the caches or the abilities of the coherence network. Partitioning an entire data object to a cache can mis-utilize the caches, not allowing for shared data to be spread across memories, and also overcommit a memory. Thus, the profile-guided method for partitioning data accesses was developed. The profile gathers statistics about affinity relationships and between memory accesses and working-set estimates of each individual access. A partition can then be made at the finer granularity of a memory operation rather than objects. This allowed for more sharing of data across PEs and caches, and improved performance in the presence of coherent data caches. The profile-driven method was able to reduce memory stall time over 90%, and 50% on average, by colocating data accesses and reducing coherence traffic.

Regardless of the technique to partition the memory, reducing the memory stall time is only half the solution, as the remaining computation must also be partitioned across the PEs. This dissertation introduced the RHOP compiler technique, which

efficiently partitioned the computation operations with a region-level scope. In addition, the technique used novel slack distribution and schedule estimation methods to improve its partitioning decisions. RHOP is able to include input from the data partitioning to determine a synergistic computation partition. The RHOP technique was shown to produce better partitions as the number of PEs increased. With the inclusion of a data partition, the overall process for fine-grain parallelism extraction was able to achieve speedups of 1.35 when increasing from a single to 2-PE processor, and 1.37 with a 4-PE processor.

The techniques presented in this dissertation combine to form an effective method for parallelism extraction for decentralized architectures. This work focused on compiler analyses, extensions and processes for parallelism extraction and management through fine-grain partitioning of data and computation. The work presented is directly applicable to the embedded-systems domain with multicluster VLIWs, but can also be applied to detecting parallelism for general-purpose multicore processors. With the abundance of decentralization in the forms of multicluster VLIW and multicore superscalar processors, compiler extensions to parallelize applications have become of growing importance for high-performance code generation.

7.2 Future Directions

There are a number of future directions in which this dissertation could be expanded and extended, including exploration of parallelized benchmarks, new architectural features, and compiler partitioning extensions.

7.2.1 Benchmarks

The current set of benchmarks examined consists of single-threaded benchmarks with varying amounts of inherent parallelism. While this benchmark set is representative of the embedded system domain, a proper examination of the applicability of fine-grain thread extraction to the general-purpose domain will need examination of explicitly threaded code. It is currently an infrastructure limitation which kept those benchmarks from being tested. When applications have already been parallelized by the programmer at a coarse-grain level, fine-grain parallelization by the compiler is still applicable on each individual thread. However, the inter-thread interaction of data must be evaluated to determine its effects on determining a good memory partition.

7.2.2 Optimal Partitioning for Small Blocks

Our compiler infrastructure currently applies a selected operation partitioning algorithm to all scheduling blocks in the input program. Previous work by Caldwell et al. [5] has shown that for small basic blocks of less than 30 nodes, FM-based partitioning can become suboptimal. In these cases, it is often preferable to use a enumeration or branch-and-bound based optimal solution. Their work focused on standard-cell layout algorithms, but the applicability of optimal solutions for small basic blocks could still be true in our operation clustering case. For some of the smallest basic blocks, a full enumeration of the problem space and a full schedule of all resulting partitions could be made. However, small basic blocks are generally

infrequently executed and make up a very small amount of the total execution time of applications.

7.2.3 Compiler Partitioning Extensions

The data and computation partitioning methods presented in this dissertation have several ways in which the could be extended. First, the choice in number of partitions to create is currently a parameter to the system, rather than a discoverable variable. As evident in many of the results, an application's ability to benefit from partitioning is highly dependent on the inherent parallelism of the code. Compiler analyses could be developed to determine an optimal number of partitions to create.

Another way in which the current techniques are limiting is the static decisions for each operation. When the resources are not fully utilized, replication of operations, rather than strict partitioning, could be beneficial. While replication increases the number of operations being executed, it could have benefits of localizing data in several PEs, and allowing for more parallel work to execute. However, this would require a significant amount of work in code generation and program restructuring to ensure correctness.

7.2.4 New Architectural Features

Currently, decentralized architectures typically have a homogeneous structure between the PEs. As processor designs evolve, heterogeneous or specialized PEs will likely become more common. Most of the techniques in this dissertation have support

for heterogeneous PEs in terms of FUs, but the handling of heterogeneous memories is not currently supported. Future decentralized architectures could specialize memories in terms of size, associativity, and power, among other characteristics.

Another way in which future decentralized architectures will undoubtedly change is in terms of their interconnect. The experiments presented in this dissertation assume either bus-based or point-to-point interconnects between the PEs. As the number of PEs scales to larger numbers, more advanced interconnections are likely necessary, such as different network topologies, varying bandwidths and protocols. These would require both the data and computation partitioners to not only partition, but also consider the geographic placement of operations on specific PEs.

BIBLIOGRAPHY

BIBLIOGRAPHY

[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, December 2001.

[2] A. Aletà, J. Codina, J. Sánchez, A. González, and D. Kaeli. Exploiting pseudo-schedules to guide data dependence graph partitioning. In *Proc. of the 11th International Conference on Parallel Architectures and Compilation Techniques*, pages 281–290, September 2002.

[3] O. Avissar, R. Barua, and D. Stewart. Heterogeneous memory management for embedded systems. In *Proc. of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 34–43, November 2001.

[4] C. Caşcaval and D. Padua. Estimating cache misses and locality using stack distances. In *Proc. of the 2003 International Conference on Supercomputing*, pages 150–159, June 2003.

[5] A. Caldwell, A. Kahng, and I Markov. Optimal partitioners and end-case placers for standard-cell layout. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(31):1304–1314, November 2000.

[6] 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, December 1992.

[7] L. Ceze, P. Montesinos, C. von Praun, and J. Torrellas. Colorama: Architectural support for data-centric synchronization. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, pages 133–134, February 2007.

[8] Y. Chen and A. Vaidenbaum. A software coherence scheme with the assistance of directories. In *Proc. of the 1991 International Conference on Supercomputing*, pages 284–294, June.

[9] H. Cheong and A. Vaidenbaum. A cache coherence scheme with fast selective invalidation. In *Proc. of the 15th Annual International Symposium on Computer Architecture*, pages 299–307, May.

[10] 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.

[11] M. Chu and S. Mahlke. Compiler-directed data partitioning for multicluster processors. In *Proc. of the 2006 International Symposium on Code Generation and Optimization*, pages 208–218, March 2006.

[12] J.M. Codina, J. Sánchez, and A. González. A unified modulo scheduling and register allocation technique for clustered processors. In *Proc. of the 10th International Conference on Parallel Architectures and Compilation Techniques*, pages 175–184, September 2001.

[13] 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.

[14] G. Desoli. Instruction assignment for clustered VLIW DSP compilers: A new approach. Technical Report HPL-98-13, Hewlett-Packard Laboratories, February 1998.

[15] A. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *Proc. of the SIGPLAN '04 Conference on Programming Language Design and Implementation*, pages 82–93, 2004.

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

[17] P. Faraboschi, G. Brown, J. A. Fisher, and G. Desoli. Lx: A technology platform for customizable VLIW embedded processing. In *Proc. of the 27th Annual International Symposium on Computer Architecture*, pages 203–213, June 2000.

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

[19] 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, December 1997.

[20] C.M. Fiduccia and R.M. Mattheyses. A linear time heuristic for improving network partitions. In *Proc. of the 19th Design Automation Conference*, pages 175–181, 1982.

[21] B. Fields, R. Bodík, and M. D. Hill. Slack: Maximizing performance under technological constraints. In *Proc. of the 29th Annual International Symposium on Computer Architecture*, pages 47–58, May 2002.

[22] 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.

[23] J. Fridman and Z. Greenfield. The Analog TigerSharc DSP Architecture. *IEEE Micro*, 20(1):66–76, 2000.

[24] 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, September 2005.

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

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

[27] E. Gibert, J. Sanchez, and A. Gonzalez. Local scheduling techniques for memory coherence in a clustered VLIW processor with a distributed data cache. In *Proc. of the 2003 International Symposium on Code Generation and Optimization*, pages 193–203, March 2003.

[28] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 151–162, New York, NY, USA, 2006. ACM Press.

[29] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14):1–6, 1996.

[30] Lance Hammond et al. Transactional memory coherence and consistency. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, page 102, June 2004.

[31] R. Hank, W. Hwu, and B. Rau. Region-based compilation: An introduction and motivation. In *Proc. of the 28th Annual International Symposium on Microarchitecture*, pages 158–168, December 1995.

[32] B. Hendrickson and R. Leland. *The Chaco User's Guide*. Sandia National Laboratories, July 1995.

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

[34] J. Hiser, S. Carr, and P. Sweany. Global register partitioning. In *Proc. of the 9th International Conference on Parallel Architectures and Compilation Techniques*, pages 13–23, October 2000.

[35] R. Ho, K. Mai, and M. Horowitz. Managing wire scaling: A circuit perspective. In *Proc. of the 2003International Interconnect Technology Conference*, pages 177–179, 2003.

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

[37] 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, February 2001.

[38] 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, September 1998.

[39] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–207, February 1970.

[40] Christoph Kessler. Optimal Integrated Code Generation for Clustered VLIW Architectures. In *Proc. of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems*, pages 102–111, June 2002.

[41] G. Krishnamurthy, E. Granston, and E. Stotzer. Affinity-based cluster assignment for unrolled loops. In *Proc. of the 2002 International Conference on Supercomputing*, pages 107–116, June 2002.

[42] V. Lapinskii. *Algorithms for Compiler-Assisted Design Space Exploration of Clustered VLIW ASIP Datapaths*. PhD thesis, University of Texas, 2001.

[43] S. Larsen, R. Rabbah, and S. Amarasinghe. Exploiting vector parallelism in software pipelined loops. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 119–129, 2005.

[44] C. Lee, M. Potkonjak, and W.H. 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.

[45] 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, October 1998.

[46] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe. Convergent scheduling. In *Proc. of the 35th Annual International Symposium on Microarchitecture*, pages 111–122, 2002.

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

[48] R. Leupers. Instruction scheduling for clustered VLIW DSPs. In *Proc. of the 9th International Conference on Parallel Architectures and Compilation Techniques*, pages 291–300, October 2000.

[49] J. Li, L. Behjat, and A. Kennings. Net cluster: A net-reduction based clustering preprocessing algorithm for partitioning and placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(4):669–679, April 2007.

[50] J. Liou and M. Palis. A new heuristic for scheduling parallel programs on multiprocessor. In *Proc. of the 7th International Conference on Parallel Architectures and Compilation Techniques*, pages 358–365, October 1998.

[51] W. Liu et al. POSH: A TLS compiler that exploits program structure. In *Proc. of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 158–167, April 2006.

[52] P. Lowney et al. The Multiflow Trace scheduling compiler. *Journal of Supercomputing*, 7(1):51–142, January 1993.

[53] R. L Matsson, J. Gecsei, D. Slutz, and I.L Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

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

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

[56] 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, December 1998.

[57] R. Rangan, N. Vachharajani, M. Vachharajani, and D. August. Decoupled software pipelining with the synchronization array. In *Proc. of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 177–188, 2004.

[58] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, November 1994.

[59] J. Sanchez and A. Gonzalez. A Locality Sensitive Multi-Module Cache with Explicit Management. In *Proc. of the 1999 International Conference on Supercomputing*, pages 51–59, June 1999.

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

[61] K. Sankaralingam et al. Exploiting ILP, TLP, and DLP using polymorphism in the TRIPS architecture. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, June 2003.

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

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

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

[65] 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.

[66] Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*, June 2004. http://focus.ti.com/lit/ug/spru189f/spru189f.pdf.

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

[68] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, September 1994.