Code and Data Partitioning for Fine-grain Parallelism

Michael L. Chu and Scott A. Mahlke

Advanced Computer Architecture Laboratory
University of Michigan, Ann Arbor, MI
{mchu, mahlke}@umich.edu

Categories and Subject Descriptors C.1.4 [*Processor Architectures*]: [Distributed Architectures]; B.3.2 [*Memory Structures*]: [Cache Memories]; D.3.4 [*Programming Languages*]: [Code generation, Optimization]

General Terms Algorithms, Experimentation, Performance

Keywords Code and data partitioning, Fine-grain parallelism, Multicore compilation

1. Introduction

The recent shift to multicore designs for mainstream processors offers the potential to improve the performance of current applications. However, converting this potential into reality is a great challenge. The programmer and/or compiler must parallelize applications to take advantage of multiple cores. Recently, a significant amount of work has focused on areas such as new programming models and ways to exploit data-level parallelism. These methods for coarse-grain parallelization can be extremely powerful in extracting large amounts of parallel work and distributing them across the cores. However, there are still 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.

This paper focuses on an alternative compiler-directed method for program parallelization by exploiting fine-grain instruction-level parallelism (ILP). Current research in interconnection networks have focused on multiple ways to increase the speed and bandwidth of communication between cores [6, 7]. Faster communication of data values between the cores can then allow for applications to take advantage of parallelization at the operation and data granularity between the cores. While coarse-grain techniques can parallelize large portions of execution, our fine-grain method can use an additional dimension to further increase performance and exploit 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 at the high level of a programming model. However, the compiler is able to take advantage of analyses of the dataflow 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. In embedded processors, centralized register files and datapaths became the cost, energy and delay bottlenecks in wide-issue designs [4]. Multicluster processors helped alleviate the scalability problem by decentralizing resources into smaller designs and grouping them together into individual processing elements (PEs) [2, 5]. These processors address the issue of fine-grain parallelism by relying on the compiler to partition operations across the program. A significant amount of previous work has focused on developing methods to partition the code across multiple clusters. The main difference between multicluster and the multicore processors of today are that multicluster generally have a shared data cache, while multicore processor 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.

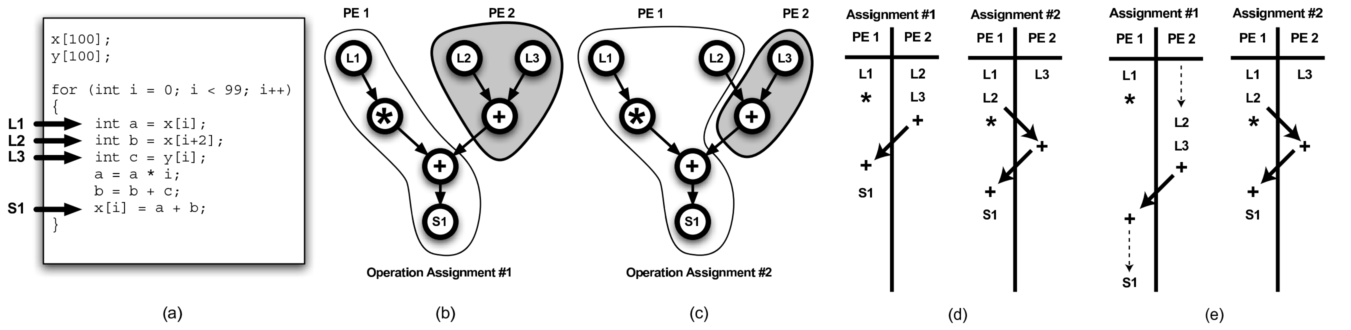
The distributed data caches requires the compiler to carefully examine the data access patterns of each individual memory operation. Analysis of the memory accesses of each operation can help to determine when individual data accesses are causing others to either hit or miss in the cache. In addition, 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 of this work is to compile to chip-multiprocessors, such as RAW, that can both exploit TLP and ILP. This work focuses on the fine-grain ILP side, where the architecture can be viewed as a multicluster VLIW with distributed/coherent L1 caches. The main problem to be solved is creating an intelligent partitioning of both the data in the application as well as the computation operations being executed. We feel a joint solution is needed to result in an efficient partitioning of the code. Thus we propose a phase-ordered approach to first partition the data, then the computation in the code.

2. Compilation Challenges

Compiling for an architecture with distributed data caches can be a challenging task. Figure 1 is an example pseudocode that illustrates some of the difficulties that can arise when the compiler

LCTES'07 June 13–16, 2007, San Diego, California, USA.
ACM 978-1-59593-632-5/07/0006.



```
x[100];
y[100];

for (int i = 0; i < 99; i++)
{
    int a = x[i];
    int b = x[i+2];
    int c = y[i];
    a = a * i;
    b = b + c;
    x[i] = a + b;
}
```

Figure 1. An illustrative example of the difficulties 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.

is partitioning the operations. In Figure 1(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 1(b) and (c).

Traditional operation partitioners [3] assign operations to clusters assuming a shared cache for the data and have a locally greedy scope for the computation. This results in poor management of the distributed caches and often produces unbalanced partitionings of the computation. A normal operation partitioner may try and produce the assignment of operations shown in Figure 1(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.

However, given a distributed data cache design, the desired PE assignment can change drastically. Looking again at Figure 1(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 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 1(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 1(d) and (e). In Figure 1(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 1(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.

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
L1 Cache	2-way associative
L1 Block Size	32 bytes
L1 Cache Size	4kB per PE
L1/L2/Main Memory Hit Latency	1/10/100 cycles
Coherence Protocol	MOESI

Table 1. Details of our simulated machine configurations

3. Experiments

Our method is a phase-ordered approach to partition memory and computation. 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. This phase uses a region-scoped algorithm, the Region-based Hierarchical Operation Partitioner [1], which partitions the operations using schedule estimates to determine a solution efficiently.

Our profile-guided data access partitioning technique was implemented as part of the Trimaran compiler infrastructure, a retargetable compiler for VLIW/EPIC processors. The machine model used was a multicluster VLIW with 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 1.

We ran our experiments on a set of DSP kernels, Mediabench and SPECint benchmarks to test programs with a broad range of inherent parallelism. For each benchmark, we evaluated the per-

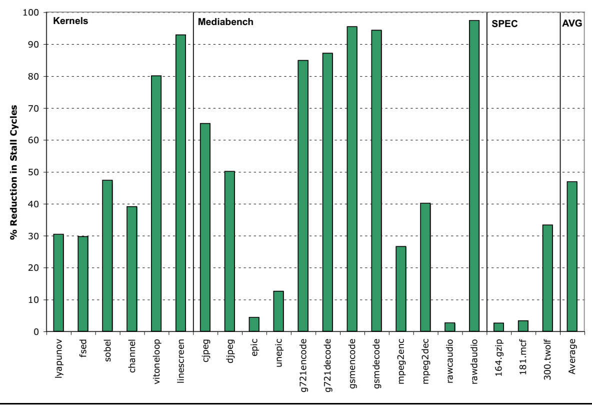


Figure 2. Reduction in stall cycles when using a profile-guided data access partitioning for a 2-PE processor

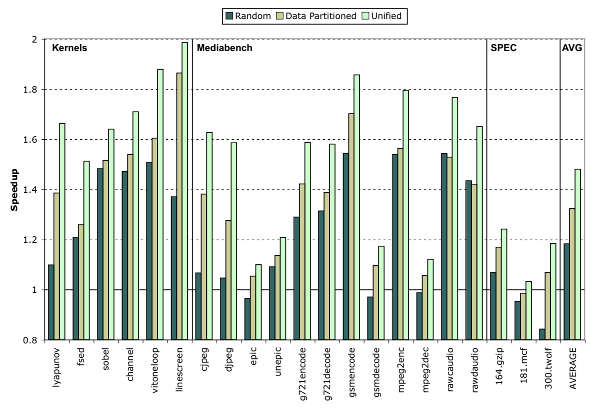


Figure 3. Speedup when using code and data partitioning across a 2-PE processor

formance of a randomly partitioned data with our profiled-guided method which prepartitions the memory accesses. Thus, our base case is a partition generation with none of our memory operation improvements, but simulated on a machine with distributed L1 data caches. 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.

Figure 2 shows the improvement in stall cycles for our profile-guided data access partitioning technique compared to the partition produced with no active data partitioning. Higher bars indicate a larger reduction in 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.

While the number of cycles due to stall is improved, total performance is affected by the sum of both stall and compute cycles. To evaluate the total performance, we measured the amount of speedup achieved when moving from a single PE processor to a 2-PE processor. In this 2-PE processor, we have double the resources, but must be carefully partition the operations and the data across the distributed caches and FUs. Thus, to achieve maximum speedup, the fine-grain parallelism must be exploited cognizant of the cache effects and the interconnection network.

Figure 3 shows the speedup for our technique. The baseline is the performance of a single-PE processor. For each benchmark, three bars are shown. The first bar shows the performance of a randomly generated data access partition. The second bar is the speedup achieved by our technique. The third bar is the speedup of a single-PE machine with double the resources. Thus, this third bar gives us an indication of the upper bound on performance.

We found that a unified, double resource machine was able to achieve an average speedup of 1.48. Our technique was able to extract an average speedup of approximately 1.32. In many cases, the randomly partitioned data shows a worse performance than the single-PE baseline. This shows that a careful placement of the data accesses is vital to producing a good fine-grain partition of the code. In one case, *181.mcf*, our partitioner produced a result slightly slower than a single-PE machine. This result came because the benchmark itself had very little parallelism, as indicated by the low speedup on the double-resource machine. In effect, our technique tried to force parallelism on an application were there was little to be found.

4. Conclusion

The recent design shift towards multicore processors has spawned a significant amount of research in the area of program parallelization. Performance gains in the future will require programmer and compiler intervention to increase the amount of parallel work possible. The future abundance of cores on a single chip offers many possibilities in ways to exploit the underlying parallel resources. In this work, we focused on a new angle for increasing the available parallelism for future multicore processors: a compiler technique to detect and exploit fine-grain parallelism. This fine-grain technique partitions code at the granularity of individual operations and data across multiple cores and caches to further increase performance. We found that many applications can be readily partitioned to multiple cores at a fine-grain level, achieving an average speedup of 1.3 on a 2-PE machine, and stall cycle reductions of 46%. Scaling to 4-PE machines, we found that our average speedup increased slightly to 1.4 with stall cycle reductions of 50%, due to lack of enough inherent parallelism.

References

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

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

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

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

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

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

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