Proceedings of the 2nd Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia (held in conjunction with IPDPS'01)

Dynamic Parallel Media Processing Using Speculative Broadcast Loop (SBL)

Jason Fritts[1] and Wayne Wolf[2]
[1]Dept. of Computer Science, Washington University, St. Louis, MO
[2]Dept. of Electrical Engineering, Princeton University, Princeton, NJ

Abstract

This paper presents the results of a study of dynamic parallel media processing using Speculative Broadcast Loop (SBL), a speculative run-time loop-level parallelization method. Due to processing regularity, multimedia applications typically contain extensive parallelism. Subword parallelism methods are commonly used to support data parallelism between independent loop iterations in inner loops, but much of the data parallelism in media processing resides in outer loops and cannot be supported with subword parallelism. Larger-scale parallel methods are needed to enable use of the full range of data parallelism in multimedia. Because static parallel compilation methods are often unable to recognize all parallelism at compile time, a run-time method is assumed for the speculative execution of potentially parallel loops. The SBL run-time method combines SIMD parallelism with large-scale speculation for supporting data parallelism in multimedia.

1. Introduction

The next generation of multimedia is rapidly approaching, in which multimedia will be described not in terms of video frames and audio channels, but instead using more advanced representations. One example is the object-oriented representation enabled by MPEG-4. The objects in MPEG-4 represent real-world objects, each with its own audio, visual, and graphical characteristics. The MPEG-4 and other advanced representations enable significantly more freedom within multimedia, allowing higher compression rates, more interactive media, and content-based processing. However, these advantages are offset by more complex processing requirements.

From a computing perspective, this new generation of multimedia will require considerably more advanced media processors. The added flexibility of these advanced representations introduces much greater computing demands and processing irregularity than seen in the first generation of digital multimedia. Adequate support for future multimedia will require processors with much higher computing capabilities than existing media processors.

To achieve the necessary throughput, future media processors will move towards higher frequencies and higher degrees of parallelism. However, because high frequency and high parallelism are counter-productive goals, we expect these processors to use more distributed architectures, such as multiprocessors, array processors, or clustered architectures, that separate the architecture into many small high-frequency processing elements. Essentially, we expect future media processors will be high-frequency *parallel media processors* (PMPs) [1].

Achieving high throughput on a PMP requires an aggressive compiler for finding sufficient parallelism. It has been found that there exists considerable parallelism in most multimedia applications [2], but the question remains as to where this parallelism resides and how it may be utilized by the compiler. It was initially believed that instruction level parallelism (ILP) would provide significant parallelism, but initial studies indicated that multimedia contains little more ILP than general-purpose applications [3]. The primary parallelism in multimedia is data parallelism, the parallelism between data elements that have little or no processing dependency between them. Subword parallelism methods (such as Intel's MMX) are often able to support the data parallelism residing in inner loops, but much of the data parallelism in multimedia is of a larger granularity, residing in the outer loops. Consequently, more traditional parallel processing support is needed for data parallelism in multimedia.

Extraction of data parallelism for parallel processing requires use of parallel compiler optimizations, as used by multiprocessors [4]. These optimizations take advantage of the parallelism found between separate loop iterations at various loop levels in the program. Those loop iterations found to be independent correspond to data parallel elements in multimedia. High degrees of parallelism can be achieved by scheduling these data parallel loops across separate processing elements in the processor. Early studies involving hand scheduling of key kernels from video applications indicate the potential for near-linear speedup [5].

One problem with parallel compiler optimizations is that recognizing parallel loops requires data dependence analysis. Unfortunately, data dependence analysis is a complex problem, so the compiler is often unable to recognize all the available loop-level parallelism. The compiler can only parallelize those loop iterations that data dependence analysis indicates are parallel. It must conservatively assume that loops having indeterminate data dependences are not parallel. In multimedia, the predictable nature of memory access typically means that potentially parallel loops are truly parallel even if they are not provably parallel. Consequently, it is desirable to have a method to optimistically assume potentially parallel loops are parallel instead of conservatively assuming they are not parallel.

This can be accomplished using a parallel media processor that supports speculative run-time parallelization. With this capability, the compiler may optimistically parallelize potentially parallel loops. Then, if the processor should execute dependent loop iterations in parallel, the speculative hardware provides a method for detecting and recovering from that misspeculation. The use of large-scale speculative execution will enable parallel media processors to most effectively utilize data parallelism, thereby maximizing performance. The method of speculative loop execution we propose is Speculative Broadcast Loop (SBL) execution.

This paper continues in Section 2, describing related research in run-time parallel processing methods. Section 3 introduces the Speculative Broadcast Loop method, describing both the compile-time and run-time requirements. Section 4 defines the evaluation environment. Section 5 examines the performance of SBL execution. Finally, Section 6 summarizes our conclusions.

2. Related Research

There does not exist any prior research that has examined run-time parallelization methods for multimedia applications, but a number of studies have been performed for general-purpose and scientific applications. As mentioned above, the problem with compile-time parallelization is that it requires data dependence analysis to guarantee the existence of parallelism. Using run-time methods provides two alternative methods to conservatively assuming loops with indeterminate parallelism are not parallel. The first is to mark the loop as indeterminate and use run-time checks to determine if the loop is parallel, and if so, execute it in parallel. The second alternative is to assume it is parallel and speculatively execute it in parallel at run-time. These two methods are often referred to as non-speculative and speculative run-time methods, respectively. For the non-speculative methods, the inspector/executor approach is usually employed, which executes a test loop before the main loop to check for any data dependence conflicts, thereby determining whether the loop is in fact parallel. Rauchwerger [6] provides a good summary of non-speculative run-time methods. As our method employs speculative run-time parallelization, we focus on the speculative run-time methods here.

Speculative run-time methods do not require the overhead of an inspector test loop. Testing of data dependences occurs simultaneously with parallel execution. Because speculative methods perform data dependence in concert with execution, these methods all provide a recovery mechanism that enables restoration of the old processor state in the event a dependence conflict occurs during execution. Three methods exist that support both fully and partially parallel loops, the Multiscalar project [8], Thread-Level Data Speculation (TLDS) [9], and Thread-Level Speculation (TLS) [10]. With each of these, there are mechanisms in either hardware and/or software for storing speculative processor state, restoring the old processor state on a misspeculation, and checking for dependence conflicts during execution. More information on these methods can be found in Fritts [11].

3. Speculative Broadcast Loop

We propose the *Speculative Broadcast Loop* (SBL) method for the speculative execution of parallel loop iterations. This new vector-like run-time method is a simplified version of the multiscalar [8] and multithreaded [9][10] methods. It combines SIMD parallelism with large-scale speculative execution for supporting data parallelism in multimedia. Unlike the multiscalar and multithreaded architectures, which provide independent control streams for separate processing units, the SBL method uses only a single control stream, which it broadcasts to each processing element for SIMD processing. The SIMD parallelism of this method is not as flexible as the multiscalar and multi- threaded implementations, but we believe this method matches well with the processing regularity in media processing, and expect it will enable similar performance levels with less hardware complexity.

Supporting only SIMD parallelism in the SBL method enables simple processor implementation via a multi-cluster architecture, a much less complicated

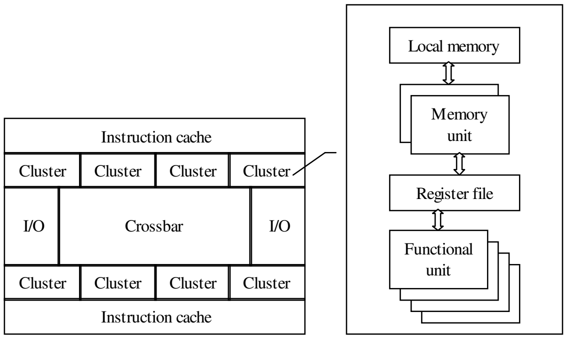

Figure 3.1 – Clustered Architecture.

architecture model than those used by the multiscalar and multithreaded architectures. The design of the multi-cluster architecture used in this project is shown in Figure 3.1. The processor contains up to 16 clusters, with 2 to 4 issue slots per cluster. Each cluster is self-contained with its own functional units, register file, and memory. A low-latency network enables communication between clusters.

The basis of the SBL run-time technique uses profiling and register dependence analysis to identify loops that are potentially parallel, and then optimistically schedules potentially parallel loop iterations across separate clusters (one iteration per cluster) in the multi-cluster architecture. During SBL execution the multi-cluster architecture simulates vector processing. The SBL "broadcasts" a single instruction control stream to each cluster so that the loop iterations are all processed in SIMD form, as shown in Figure 3.2.

To enable SIMD execution of both outer loops and loops with complex control (MIMD) structures, parallel loops are scheduled using *Multi-Level If-Conversion* (MLIC) to eliminate all unnecessary branches, and speculative hardware is provided for recovering from instruction flow deviations between parallel iterations. The speculative hardware also enables recovery for iterations that have memory conflicts (i.e. loops that are partially parallel or not parallel) and for iterations that have executed beyond the bounds of the loop (i.e. overshot the loop termination condition). A limitation of this method is that its SIMD nature prohibits parallelism across function boundaries. Consequently, while the SBL method supports both fully parallel and partially parallel loops, it cannot support arbitrary-sized tasks like the multiscalar and multithreaded architectures.

There are two major aspects to the SBL run-time parallelization method. These two aspects are not inter-dependent and could be used individually, but together they provide the greatest benefit:

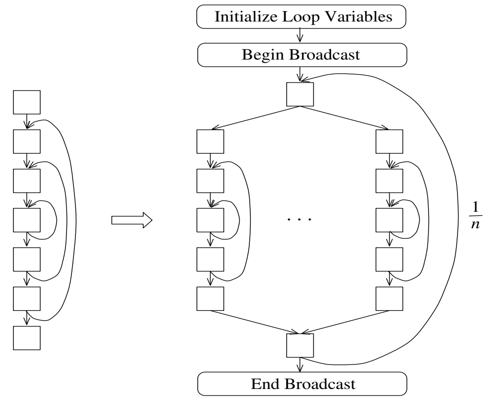

Figure 3.2 – SIMD parallelism model for broadcasting and speculatively executing parallel loops on a n-issue cluster architecture; broadcast loop back-edge is now taken only 1/n times.

1. Loop broadcast and multi-level loop scheduling for SIMD parallelism across a multi-cluster architecture.
2. Hardware extensions for supporting large-scale speculative execution of parallel loop iterations.

These two aspects, as well as the corresponding architecture and compiler implications, will be discussed in the remainder of this section. Discussion of the Speculative Broadcast Loop method will first examine how profiling and register dependence analysis are used for finding potentially parallel loops. Then we shall examine how Multi-Level If-Conversion is used for scheduling parallel loops for broadcast on the multi-cluster architecture. Finally, large-scale speculative execution and the resulting hardware modifications are discussed. More detailed information on SBL is available in Fritts [11].

3.1. Finding Parallel Loops

Profiling and register dependence analysis are employed to determine which loops are potential candidates for loop parallelization. An alternative for finding parallel loops would be to use parallel compiler methods such as those discussed in the prior section. However, profiling can provide exact memory access addresses for performing dependence analysis, while traditional data dependence analysis can only make estimates. Consequently, profiling is the method currently used by the SBL method for finding parallel loops.

To generate loop profile information, the program trace is instrumented with additional information indicating the start of each loop, the beginning of subsequent iterations for each loop, and the end of each loop. Then the profiler is augmented with data structures that record the loop statistics as profiling is performed. Because loops are executed in a nested fashion, a loop execution stack is used for correctly organizing the data structures within nested loops. These data structures keep track of every byte of memory accessed in the iteration currently executing.

Once a loop completes execution, the memory access information for each iteration is compared to determine the number of memory conflicts per iteration, which will be zero if the loop is potentially parallel. The completed loop is popped off the loop execution stack, and the aggregate memory access information for all iterations are combined into the memory information for the current iteration of the next lower loop on the stack. This way the memory information can be maintained at all loop levels.

While the loop profile statistics define the memory data dependences in each loop, it is also necessary to evaluate the register-based data dependences in each loop. This can be accomplished using static code information after profiling is complete. The existence of cross-iteration dependences from register operands can easily be determined using knowledge of the induction variables and live-out sets for each loop. This is accomplished using traditional liveness and induction variable analysis. If there are no cross-iteration register dependences, then the loop is potentially parallelizable. When both the memory dependences and register dependences indicate a potentially parallel loop, the loop may be a candidate for SBL execution.

3.2. Scheduling Loops for Broadcast

After defining the set of candidate loops, it is necessary to select the appropriate loops and schedule them for SBL loop parallelization. Selection of loops can be done according to any heuristic as long as two basic rules are followed. First, only one loop level can be selected for parallel execution. If a multi-level loop is parallel at multiple levels, only one level may be selected for loop broadcast. Second, because this method uses SIMD processing, loops containing function calls cannot be parallelized. The processor cannot guarantee the same flow of control within parallel function calls, so SBL across function calls is not allowed. Outside of these criteria, any heuristic may be used to select the loops for SBL. We found a reasonable heuristic to be one that favors outer loops levels over inner loop levels as long as profiling indicates an average of 2+ iterations for that loop.

After selecting the loops to parallelize, it is necessary to schedule them for maximum SIMD parallelism. This essentially entails removing all unnecessary branches. We propose the Multi-Level If-Conversion (MLIC) method for scheduling broadcast loops. In loops that contain multiple control paths, this method uses predication to combine all control paths into a single control path to enable SIMD processing. We shall first describe how MLIC works on inner loops, and then extend that for scheduling of multi-level loops. For an inner loop, combining multiple paths of control flow into a single control path proceeds as follows:

1. Combine all loop back-edges into a single back-edge.
2. Combine all loop exits into a single loop exit; insert extra branches outside the loop body if there are multiple exit destinations.
3. Use if-conversion on the resulting diamond region to form a single control path.

The first two steps essentially convert the loop body into a diamond region with only one entry pointand one exit point. The third step then predicates the diamond region into a single control path using if-conversion.

When the loop being parallelized is an outer loop, the situation is a bit more complicated. In this case, it is no longer possible to eliminate all intra-loop branches. Some of these branches will be associated with loop back-edges and loop bypass branches for loops nested within the parallel loop. However, if-conversion can still be used on individual regions within the loop to eliminate all unnecessary branches. A recursive pseudo-code procedure for accomplishing this is given in Figure 3.3, and an example outer loop SBL compilation is shown in Figure 3.4.

```
combine_loop_exits (main_loop) {

  if (main_loop has child loops) {
    for each (child_loop) {
      combine_loop_exits (child_loop);
      combine_pre_loop_exits (child_loop);
    }
    combine_post_loop_exits (    child_loop);
  }
  else
    combine_inner_loop_exits (child_loop);
}
```

Figure 3.3 – Recursive procedure for combining loop exits in outer loops.

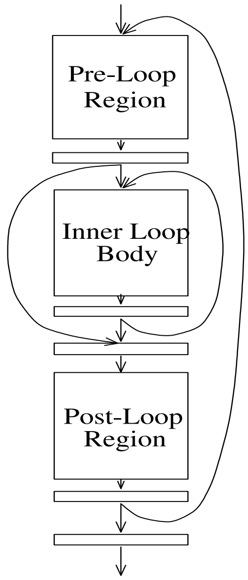

Figure 3.4 – Multi-level if-conversion an outer loops eliminates all unnecessary branches.

The above procedure for performing Multi-Level If-Conversion on a broadcast loop splits a multi-level loop into nested loop regions, pre-loop regions, and one post-loop region. Nested loop regions are recursively processed with the given procedure until an inner loop is found. The inner loop is processed according to the method defined earlier. For each nested loop there is a section of code prior to the loop, which we refer to as the pre-loop region. This region is if-converted in a similar manner to inner loop regions. For branches that branch out of the pre-loop region, but do not jump to the succeeding nested loop, a branch bypass path is taken around the nested loop to the next successive loop region.

Because branches still exist in SBL parallelized outer loops, there is the potential for separate parallel iterations to have instruction control flow deviations when nested loops branch in different directions. In such events, the speculative execution support comes into play and squashes the deviating loop iterations. This will be discussed in greater detail shortly.

After performing Multi-Level If-Conversion on the parallel loops, it is also necessary to insert the operations for setting and resetting broadcast mode, performing synchronization, and initializing loop variables. Towards this end, *begin broadcast* and *end broadcast* statements are used to put the processor into broadcast mode and back into normal mode, respectively. These are to be placed immediately before and after the beginning and ending of a parallel loop. Synchronization operations are also needed for supporting speculative execution. These operations act as checkpoints. At each checkpoint, the processor state is saved to provide a recovery point in case a misspeculation occurs. At the beginning of the loop, the *begin broadcast* operation can double as a synchronization operation, but synchronization is also needed at both the end of the loop and between groups of parallel iterations, so the *end broadcast* operation is not sufficient. A *checkpoint* operation needs to be placed in the parallel loop's back-edge/exit basic block so that synchronization occurs on both the loop back-edge and the loop exit.

Initialization operations are also needed to prepare all the variables for parallel loop execution. There are two types of initialization operations. The first type are copy operations for variables in the loop's live-in set. Each loop needs access to all variables in the live-in set, so they must be copied to each cluster. The second type of operation initializes the loop induction variables for each iteration.

After Multi-Level If-Conversion and initialization of loop variables, the program code is ready for parallel execution.

3.3. Speculative Hardware Support

Because static dependence analysis was not used to guarantee that there are no data dependences in parallel loops, it is necessary to assume that memory conflicts will occur. The loop memory independence profiling helps determine whether loops have data dependences under a given input set, but this does not definitively determine loop memory independence. Fortunately, the processing regularity in multimedia enables the profiling results to be relatively accurate, so by only choosing loops for parallelization whose profiles indicate no memory conflicts, memory conflicts will typically occur infrequently. Even so, it is necessary to provide a method of recovery when a memory conflict between loop iterations does occur.

In addition to supporting speculative execution in the shadow of potential memory conflicts, it is also necessary to support control flow speculation. When scheduling an outer loop for SBL execution, there is the potential of control flow deviations from branches for loops nested within the outer loop. A method for squashing the deviating outer loop iterations and recovering from the misspeculation is necessary.

Finally, it is also desirable to support parallelism on loops, such as while loops, which have an indeterminate number of loop iterations. This can only be achieved by speculatively executing as many parallel iterations as possible and then squashing and recovering from those iterations that exceeded the loop bounds. Summarized, these three areas of speculation are:

1. Memory Independence Speculation
2. Control Flow Speculation
3. Parallel Loop Iteration Speculation

Of these three, memory independence speculation is the most difficult to support. The other two areas of speculation are primarily covered by the speculation and recovery requirements of Memory Independence Speculation, so they can be supported with little or no additional hardware.

During SBL execution, three items are necessary. 1) A method for storing speculative state during SBL execution, 2) A means for storing the processor state prior to beginning speculative execution, and 3) A mechanism for dynamically checking data dependences to determine if a memory conflict occurs. The data checker (item 3) is used to determine if a memory conflict occurs. If a memory conflict occurs, or one of the other two forms of misspeculation occurs, the processor needs to invalidate all the speculative processor state (item 2), and backup to the last valid processor state (item 1). Conversely, if speculative execution proceeds without a misspeculation, then at the next checkpoint or at the end of speculative execution, the current speculative state becomes valid and can be saved over the last valid processor state.

The first of these, storing speculative state during SBL execution, requires a significant amount of speculative storage space. Performing SBL execution on an outer loop could easily entail hundreds of speculative memory accesses. One common method for providing speculative state is maintaining speculative state in the register file, and holding a backup of the last valid processor state in a backup register file. To support such large-scale speculation, however, we need additional memory storage. As proposed by the multithreaded architectures methods [9][10], we use the L1 cache for maintaining speculative state.

Since the register file and L1 cache are being used to maintain speculative state, the SBL method uses a backup register file and the L2 cache to store the backup processor state. In order to avoid having to copy all the data in the L1 cache to the L2 cache and invalidate all the L1 cache lines at the beginning of SBL execution, this method adds an extra bit to each L1 cache line for indicating speculative data. Consequently, valid L1 cache lines are only copied to the L2 cache if a speculative access is made to that cache line and that line is currently marked dirty and non-speculative.

The full speculative model for the SBL method therefore includes restorable register files and checkpointing as well as a speculative L1 data cache.

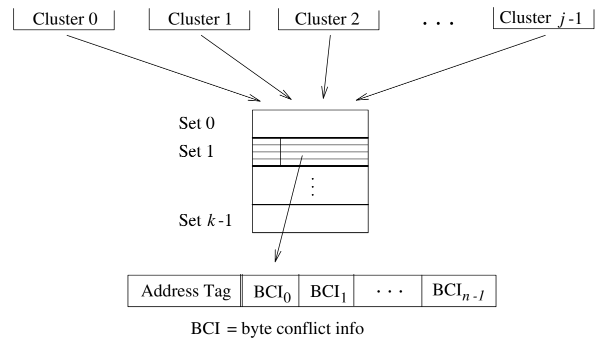

Figure 3.5 – Global loop memory conflict (LMC) cache design.

The final requirement for speculative execution is a test mechanism for checking if memory conflicts exist. In the multiscalar and multithreaded projects they build data conflict checking into their cache coherence protocol [8][9][10]. While this method is also feasible in the multi-cluster architecture, we propose using a separate cache, the Loop Memory Conflict (LMC) cache, to perform conflict checking. The job of the LMC cache is to keep track of all memory accesses in each parallel loop iteration and perform cycle-by-cycle checking for memory conflicts. Performing a continuous evaluation of memory conflicts eliminates nearly all post-loop memory conflict checking overhead and also enables early notification of misspeculations. Early notification prevents misspeculated loop iterations from continuing to access memory locations, and so avoids unnecessary memory stall penalties.

Figure 3.5 displays the design of a global LMC cache. A distributed design is also possible, and is discussed in Fritts [11]. The BCI field maintains the byte conflict information for each byte that is written during SBL execution. The BCI field contains a bit for each cluster (i.e. a bit for each iteration) available for SBL execution. So for a 4-cluster machine there are 4 bits per byte, while an 8-cluster machine would require 8 bits per byte. Our initial results indicate a LMC cache that monitors around 8 KB of speculative data is likely sufficient for most applications.

4. Evaluation Environment

Accurate evaluation of the SBL method in media processing requires a benchmark suite representative of the multimedia industry, and an aggressive compilation and simulation environment. This architecture style evaluation uses an augmented version of the MediaBench benchmark suite defined by Lee, et al. [12] at UCLA. This benchmark provides applications covering six major areas of media processing: video, graphics, audio, images, speech, and security. MediaBench was augmented with the MPEG-4 and H.263 video applications to help make the benchmark more representative of the current and future multimedia industry.

The compilation and simulation tools for this study were provided by the IMPACT compiler, produced by Wen-mei Hwu's group at UIUC [13]. The IMPACT environment includes an aggressive ILP compiler and a trace-driven simulator. The IMPACT compiler supports many aggressive compiler optimizations including procedure inlining, loop unrolling, speculation, and predication. However, only the classical ILP compilation method was used during this initial investigation. The IMPACT simulator is a parameterizable trace-driven simulator that enables both statistical and cycle-accurate simulation of a variety of uniprocessor architecture models, including VLIW, in-order superscalar, and out-of-order superscalar datapaths. For evaluation of the SBL method, we also recently expanded the simulator to model multi-cluster architectures. As described above in Section 3, the multi-cluster architecture provides each cluster with its own functional units, register file, and memory.

5. Experiments

To examine the effectiveness of the SBL method for multimedia, we compiled and simulated the MediaBench benchmark suite under the IMPACT environment on a multi-cluster architecture while varying the number of clusters. This experiment evaluates the performance of multi-cluster architectures with VLIW, in-order superscalar, and out-of-order superscalar datapaths. The results measure the speedup as the ratio of the performance of multi-cluster architecture versus a single-cluster (i.e. non-clustered) architecture, which has resources equal to one cluster on the multi-cluster architecture.

The base processor model for these experiments is a 4-issue single-cluster processor targeting the frequency range from 500 MHz to 1 GHz, and uses instruction latencies modeled after the Alpha 21264. The functional resources of the base processor include 4 ALUs, 2 memory units, 1 shifter, 1 multiplier, 1 float-point unit, and 1 branch unit. Both the multi-cluster and single-cluster architectures assume an infinite cache model for this initial investigation.

The average results across all of the MediaBench applications are given in Figure 5.1 and Figure 5.2. Figure 5.1 shows the average number of loop iterations the SBL method was able to execute within parallel loop regions. The *ideal* number of iterations when executing in broadcast mode is equivalent to the number of clusters in the multi-cluster architecture. The *perfect* results define the average number of loop iterations that could be executed if none of the loops were squashed (i.e. no memory conflicts, no instruction flow deviations, etc.). The *true* results indicate the average number of loop iterations that are actually executed when accounting for squashed loops that had to be re-executed. Overall, it can be seen that the parallelism results are quite good. For up to 8 clusters, the true results are within 80% of ideal performance and 90% of perfect performance.

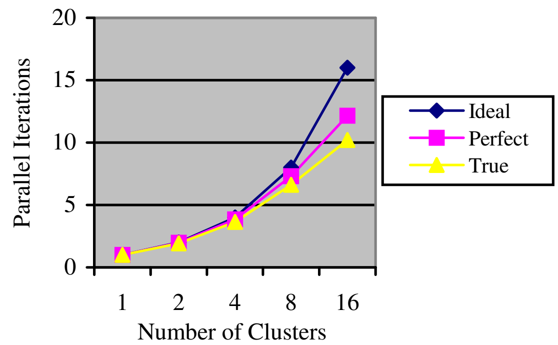

Figure 5.1 – Average number of loops executed in parallel by SBL method.

As shown in Figure 5.2, the performance of the SBL method on full applications is not quite as good. The SBL method was not able to find large enough amounts of parallelism to achieve more than 2x speedup on average. We examined the loop profiling statistics to determine what was causing the problem. As shown in Figure 5.3, we found the overall parallelism in these applications averages about 40-50%, and only a few of the applications exceed 50%

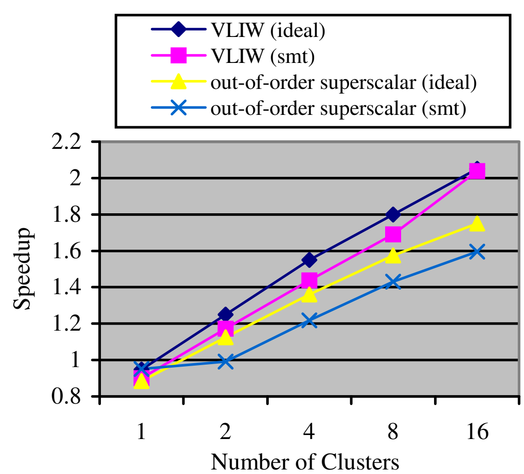

Figure 5.2 – Average performance of SBL method on MediaBench.

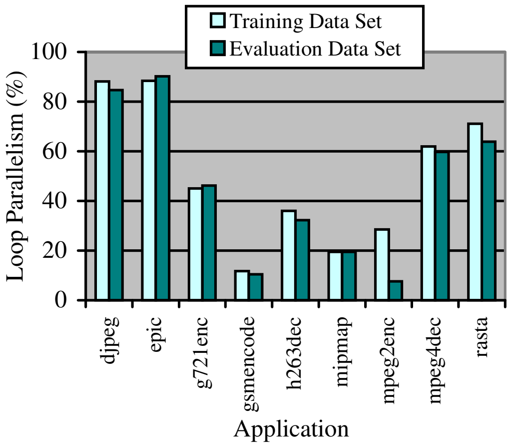

Figure 5.3 – Maximum loop parallelism according to profiling statistics.

parallelism. Those applications that did have extensive parallelism however, did show excellent SBL performance. The *epic* and *djpeg* benchmarks, which both have nearly 90% parallelism, demonstrated speedups of nearly 6x and 3x, respectively. But overall, we expected that the parallelism would be higher in most applications.

An additional experiment was performed to determine if parallelism could be improved using parallel compilation methods. By manually applying parallel compiler optimizations to critical code sections, we found that the parallelism of many of the applications could be improved substantially (nearly tripled). This insight is leading us to examine the use of a parallelizing compiler, such as SUIF, to first applying parallel compilation before performing SBL execution. Results combining SBL and a parallelizing compiler will be reported in a future publication.

6. Conclusions

Overall, we can conclude that the Speculative Broadcast Loop (SBL) method worked very well on the MediaBench applications. When parallelism was found to be available, the SBL method enabled performance within 80% of ideal performance with up to 8 clusters. The problem arose from the fact that profiling and register dependence analysis were unable to recognize significant degrees of parallelism. The typical parallelism was only about 40-50%, so the average speedup was only about 2x on 8 and 16-cluster processors. While this is lower than expected, this performance is quite good considering that the overall IPC when combined with ILP well exceeds the typical IPC from using just ILP. Additionally, the SBL method was able to display speedups of nearly 6x on applications that did exhibit significant degrees of parallelism. And the manual application of parallel compilation methods enables a significant increase in parallelism in many applications, nearly tripling performance of the SBL method. Consequently, we expect that combining SBL execution with a parallel compiler will yield exceptional results.

References

[1] Jason Fritts, Zhao Wu, and Wayne Wolf, "Parallel Media Processors for the Billion-Transistor Era," Intl. Conference on Parallel Processing, Sept. 1999.

[2] Heng Liao and Andrew Wolfe, "Avaliable Parallelism in Video Applications," 30th Annual International Symposium on Microarchitecture, Dec. 1997.

[3] J. Fritts, W. Wolf, and B. Liu, "Understanding multimedia application characteristics for designing programmable media processors," SPIE Photonics West, Media Processors '99, Jan. 1999, pp. 2-13.

[4] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua, "Automatic Program Parallelization," Proceedings of the IEEE, vol. 81, no. 2, February 1993, pp. 211-243.

[5] A. Wolfe, J. Fritts, S. Dutta, and E. S. T. Fernandes, "Datapath Design for a VLIW Video Signal Processor," 3rd Intl. Symposium on High-Performance Computer Architecture (HPCA), Jan. 1997.

[6] Lawrence Rauchwerger, "Run-Time Parallelization: It's Time Has Come," Journal of Parallel Computing, Special Issue on Languages and Compilers for Parallel Computers, 24(3-4), 1998.

[7] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, and P. Tu, "Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing," IEEE Transactions on Parallel and Distributed Technology, vol. 2, no. 3, pp. 37-47, 1994.

[8] Manoj Franklin, "The Multiscalar Architecture," Ph.D. Thesis, Department of Computer Science, University of Wisconsin at Madison, 1993.

[9] J. G. Steffan and T. C. Mowry, "The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization," 4th Intl. Symposium on High-Performance Computer Architecture, Feb. 1998.

[10] J. Oplinger, D. Heine, S. Liao, B. A. Nayfeh, M. S. Lam, and K. Olukotun, "Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor," Computer Systems Lab Technical Report CSL-TR-97-715, Stanford Univ., Feb. 1997.

[11] J. Fritts, "Architecture and Compiler Design Issues in Programmable Media Processors," Ph.D. Thesis, Dept. of Electrical Engineering, Princeton Univ., 2000.

[12] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia Communications Systems," 30th Intl. Symposium on Microarchitecture, Dec. 1997.

[13] IMPACT compiler group at University of Illinois at Urbana Champaign: http://www.crhc.uiuc.edu/Impact