Journal of Universal Computer Science, vol. 6, no. 10 (2000), 908-927
submitted: 8/2/00, accepted: 8/9/00, appeared: 28/10/00

On Dynamic Speculative Thread Partitioning and the MEM-slicing Algorithm

Lucian Codrescu
Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332-0250
lucian@ece.gatech.edu

D. Scott Wills
Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332-0250
scott.wills@ece.gatech.edu

Abstract: A dynamic speculative multithreaded processor automatically extracts thread level parallelism from sequential binary applications without software support. The hardware is responsible for partitioning the program into threads and managing inter-thread dependencies. Current published dynamic thread partitioning algorithms work by detecting loops, procedures, or partitioning at fixed intervals. Research has thus far examined these algorithms in isolation from one another. This paper makes two contributions. First, it quantitatively compares different dynamic partitioning algorithms in the context of a fixed microarchitecture. The architecture is a single-chip shared memory multiprocessor enhanced to allow thread and value speculation. Second, this paper presents a new dynamic partitioning algorithm called MEM-slicing. Insights into the development and operation of this algorithm are presented. The technique is particularly suited to irregular, non-numeric programs, and greatly outperforms other algorithms in this domain. MEM-slicing is shown to be an important tool to enable the automatic parallelization of irregular binary applications. Over SPECint95, an average speedup of 3.4 is achieved on 8 processors.

Keywords: analysis and design aids, control structure performance, architectures, multiprocessor
Categories: B.1.2, C.1.2

1 Introduction

Microprocessor architecture research is increasingly looking at multithreading to increase parallelism, tolerate latency, and reduce design complexity. A stigma of traditional multithreading is poor performance on legacy sequential binaries. A new class of dynamic multithreaded architectures is emerging to meet this need [2][5][14][18]. These designs automatically extract thread-level parallelism from binary applications. A defining feature of these architectures is how threads are extracted. Static thread partitioning is a difficult problem, as the compiler must balance the often-conflicting needs of data flow, control flow, and load balance. Dynamic partitioning algorithms do not have the benefit of compile time information,

and must look for clues in the dynamic execution. This paper begins with an example from *compress95* that illustrates some of the intricacies of the partitioning problem.

Current dynamic partitioning algorithms [2][14][18] attack either loops, procedures, or cache line boundaries. Thus far, researchers in this area have presented individual algorithms only in the context of different architectures, making comparisons between algorithms difficult. In this paper, current dynamic partitioning algorithms are quantitatively compared in the context of the Atlas chip-multiprocessor [5]. This design starts with a conventional shared memory multiprocessor. Support for thread speculation (multiscalar execution) and aggressive inter-thread data value prediction are added.

The paper then introduces a new dynamic partitioning algorithm, called MEM-slicing. The basic algorithm is presented along with an exploration into several important design variables. SPECint95 benchmarks are executed on a detailed simulator and extensive performance evaluations are performed. The MEM-slicing dynamic partitioning algorithm is shown to be an important tool to extract thread-level parallelism. Using unmodified sequential SPECint95 binaries, speedup due to thread parallelism averages 3.4 on 8 processors.

2 Related work

Speculative multithreading was introduced by the Multiscalar [10][20] architecture. This design uses the compiler to divide the program into threads and schedule inter-thread register communication. Hydra [17], Stampede [21], RAW [27], Iacoma [12], SPSM[9], Superthreaded[22], and Multithreaded Decoupled[8] also perform speculative multithreading. All of these architectures perform partitioning in software and require source code recompilation.

While work has been done in static thread partitioning for speculative multithreaded processors [25][26], very little work exists on dynamic partitioning algorithms. Current dynamic partitioning algorithms [2][14][18] attack either loops, procedures, or cache line boundaries. Loops are a natural first place to look, as parallelizing compilers have a long history of loop parallelization. Marcuello, Gonzalez and Tubella present a microarchitecture that dynamically recognizes loops and attempts to parallelize the loop iterations [14][23]. Stride based data value prediction is used to speculate inter-thread data dependencies. The trace processor [18] partitions threads on trace cache line boundaries. The cache lines are executed on separate execution cores with inter-trace dependencies being both speculated and synchronized via a global register file and global data cache. The dynamic multithreaded processor (DMT) [2] recognizes loops and procedures. Threads are dynamically forked after procedures (the code after a procedure is executed in parallel with the procedure), and after-loops (the code after a loop is executed in parallel with the loop). The execution architecture is a modified Simultaneous MultiThreaded (SMT) [24] design. It includes mechanisms to speculate on, to track, and to recover data value predictions.

This paper is divided into seven sections. Section 3 briefly describes the Atlas chip-multiprocessor. Section 4 discusses the problem of partitioning for a speculative multiprocessor, and includes an example from SPECint95. Section 5 presents a quantitative comparison of current partitioning algorithms in the context of the Atlas

multiprocessor. Section 6 introduces and explores the MEM-slicing partitioning algorithm. Section 7 presents a summary and conclusion.

3 Atlas chip-multiprocessor

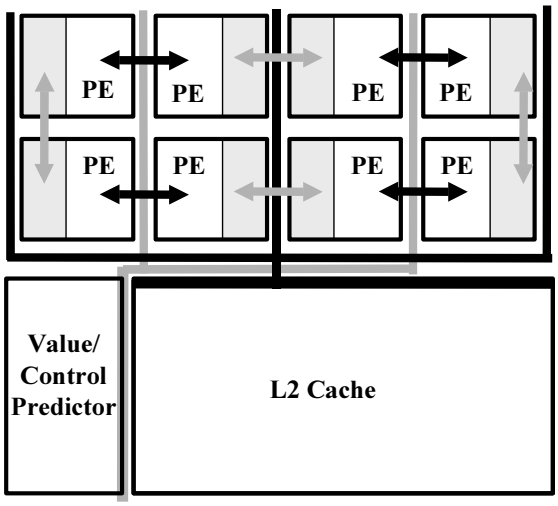

Figure 1: Atlas Architecture Block Diagram

The partitioning work presented in this paper assumes a fixed configuration of the Atlas chip-multiprocessor [5]. This section provides a brief overview of the architecture. Interested readers should refer to [5] for details. The general philosophy of this design is to start with a single-chip multiprocessor and add mechanisms necessary to support thread and value speculation.

Figure 1 shows a top-level view of the architecture. Eight processors are integrated together with global L2 cache, and a global control and value predictor. The processors communicate with the global structures through shared buses. Processors communicate with one another via a bidirectional, pipelined ring interconnect. Parallel operation is achieved on sequential binaries by dynamically partitioning a program into threads, and then executing the threads speculatively, following the multiscalar execution model. One processor in the ring is non-speculative. The thread on this processor runs normally, committing STORE results to the L2 cache. Control speculations decide which threads to allocate to subsequent processors. Threads are issued and retired in-order. Speculative threads buffer state until they become non-speculative, at which time they commit results.

Inter-thread data dependencies are handled using two mechanisms, data dependence speculation [15] and data value prediction [13]. Live register inputs are always value predicted. A hybrid global value predictor makes aggressive, 2-level, correlated value predictions [6]. All value predictions are tracked within the processors via a small fully associative queue, called the Active Ins Queue. For memory dependencies, a dependency predictor is first consulted to decide if a communicating STORE is likely. If it is not likely, the processor LOADs the value from cache and proceeds. The primary data cache is modified to track these

speculations, in a manner similar to other designs [11][12][17][21]. Recovery from this type of misspeculation is achieved by squashing the thread and restarting. Should the dependency predictor indicate a communicating STORE is likely, the data value predictor is accessed. These predictions are tracked in a similar manner to register value predictions.

The main issue queue in each processor is modified to provide fine-grained value misspeculation recovery. If a register or memory data value prediction is incorrect, only instructions dependent on the incorrect value re-execute, in a manner similar to the trace queues described in [2]. Table 1 shows the processor configuration that will be used for all subsequent simulations.

The simulator is built on the Simplescalar toolkit [1], which uses a MIPS-like ISA. It is execution-driven, fully modeling execution under misspeculation conditions. SPECint95 benchmarks are compiled with gcc 2.6.1 using full optimizations. All simulations are done using the train input sets. Benchmarks are simulated for 200 million instructions, or until they complete, whichever is first.

Processors	8
Value/Control Predictor	128Kb HLG[6]
Speculative Data Cache	32Kb,2-way SA, 4 cycle miss penalty to L2
Active In Queue	16 entry Fully Associative
Dependency Predictor	8Kbyte
Write Buffer	64 entry Fully Associative
Processor Configuration	Single issue, in-order, pipelined (5 stage pipeline)
Issue (recovery) Queue Size	64 entry
Instruction Cache	Perfect
L2 cache	Perfect
All Global Communication	4 cycles, pipelined

Table 1: Simulated Atlas configuration

4 Partitioning for thread speculation

In order to achieve speedup due to parallel execution, the application program must be divided among the processors. In a conventional multiprocessor, a compiler typically transforms loops for parallel execution. Different iterations of the loop are executed in parallel on different processors. The work done on a processor is generally called a thread, which is defined as any contiguous execution sequence. A thread could be a loop iteration, a basic block, or a procedure, for example. Historically, partitioning a program into threads meant finding guaranteed independent work for the processors. Speculative multiprocessors relax dependency restrictions in data and/or control. The processors speculate on inter-thread dependencies and then later perform corrective procedures in the case of a misprediction. Critical to the success of speculative multithreading is extracting threads that are load balanced, data predictable, and control predictable.

The size of a thread is defined as the number of dynamic instructions in the thread. It is very important that all threads are close to the same size to handle load balance. Otherwise, processors with small threads will idle while processors with large threads

complete. Thread coverage is a related issue. It is important that the entire program be divided into threads. If only some portions are threaded, execution into the unthreaded code will cause severe load balance problems.

Thread data predictability refers to the number and predictability of live data inputs into a speculative thread. For fully parallel execution, there should be no live data inputs into the thread. This is often not possible, in which case it is desirable to have the minimum number of short, unpredictable inputs. If a data input is predictable, it can be removed with data value prediction. Short, unpredictable inter-thread data dependencies should be avoided, as these will cause the consuming thread to wait for the producing thread, effectively sequentializing thread execution.

Control predictability refers to the relative difficulty in determining which thread follows the current thread. Depending on how threads are partitioned, there can potentially be many possible following threads. Threads can also be partitioned to take advantage of control convergence, by encapsulating hard-to-predict branches within a thread. Large amounts of work may be discarded as the result of incorrect thread control predictions.

The ideal thread example is a parallel loop iteration. Each loop iteration is typically the same size as all other loop iterations, has no live inputs, and control proceeds regularly from one iteration to the next. Unfortunately, many important applications (and benchmarks) do not have parallel loops.

4.1 Compress example

Finding good threads in a sequential program often involves trading off load balance, control predictability, and data predictability. This is illustrated with an example from the SPECint95 benchmark *compress95*.

```
while (c = getchar()) {
      ....
      ....     = ent
      ....
if ( ... ) {
    ent = fcode(i)
    ...
}
    ....
nomatch:
    output();
      ....
      ....
    ent = c
     ....
}
```

Figure 2: Compress main loop

The main compress loop is outlined in Figure 2. Each iteration of the loop begins by reading in a character to compress. The loop iteration then follows complex and data dependent control flow. There are numerous iteration carried true data

dependencies, some of which are inherently unpredictable. A particularly hard-to-predict and restrictive data dependency is the **ent** variable, shown in boldface. Occasionally, the loop body makes a call to the *output*() procedure to dump compressed characters.

In earlier work, the authors presented a methodology to study the data flow behavior of thread partitions in sequential programs [4]. Figure 3 shows the results of this analysis for two iterations of the compress loop. The x-axis is time measured in dynamic instructions. It shows two iterations of the compress loop from the beginning of the first iteration to the end of the second iteration. On the y-axis, the strength of data dependencies is measured. A low number indicates few hard-to-predict dependencies crossing a thread boundary defined by the dynamic instruction on the x-axis. A high number indicates that the data crossing this dynamic instruction is shorter-lived and/or hard-to-predict. There are two valleys of interest in each iteration of the compress loop. The first is the beginning of the compress loop, and the second is a call to the output() procedure. This graph indicates that those two places make good thread partitions as far as data flow is concerned, while the work between these valleys is more closely related in data flow.

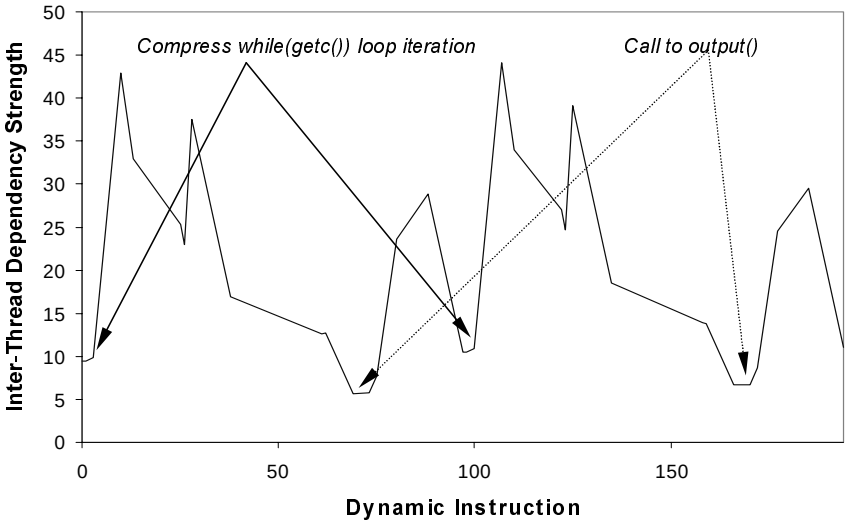

Figure 3: Compress dynamic data flow analysis

One way to partition this code is to thread the loop body. This approach is illustrated in Figure 4.

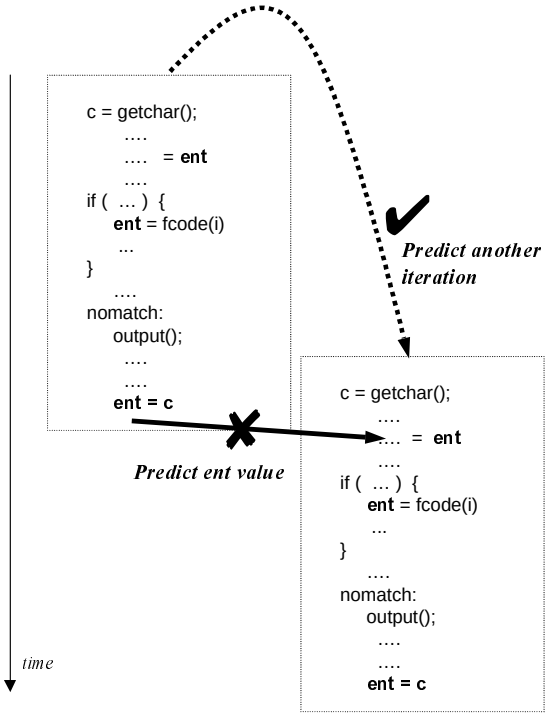

Figure 4: Loop threading compress

Control predictability is well satisfied because the loop consecutively iterates many times. Data predictability suffers because the **ent** variable forms a hard-to-predict, short iteration-carried dependency. This variable has the effect of sequentializing iterations. Also, thread size can vary considerably depending on intra-thread control flow, and as a result, load balance suffers. Even with the limitations imposed by the **ent** variable, some speedup is seen with this approach. The call to *getchar*() can be overlapped with the previous loop iteration.

Another threading approach is to thread the loop body and the *output*() procedure. This procedure has only easy-to-predict input dependencies, making it a good candidate for speculative multithreading. Figure 5 shows the results of this approach. The data flow properties of this partitioning are much better than before, as the *output*() procedure execution can be overlapped with the compress loop iteration. However, this approach introduces a control flow problem. The *output*() procedure is not called every iteration of the compress loop. It is called approximately every 2-5 iterations, but depends on the program input and can vary considerably. For this reason, it is difficult to predict the thread that follows the compress iteration thread. Sometimes another iteration thread follows, sometimes the *output*() procedure. Load balance is also affected because the size of the *output*() procedure does not match the size of the compress iteration thread.

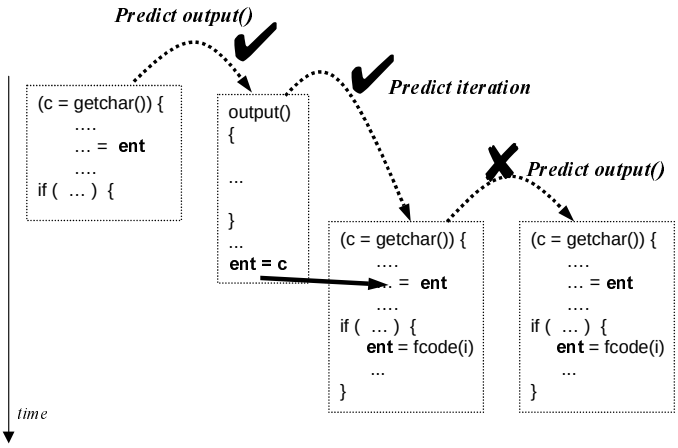

Figure 5: Threading the output() procedure

This example was presented to illustrate the delicate nature of the partitioning problem. Data predictability, control predictability, and load balance can often be traded against one another. Partitioning is a known NP-hard problem [19], and solutions are generally heuristic based. Partitioning a program dynamically is more difficult, as the run-time hardware does not have the benefit of compile-time information.

5 Current approaches

This section takes a detailed look at current approaches to dynamic partitioning in the context of the Atlas chip-multiprocessor.

5.1 Loops

Marcuello, Gonzalez, and Tubella present a microarchitecture that dynamically parallelizes loop iterations [14][23]. Stride based data value prediction is used to break iteration-carried true data dependencies. They report good performance for loop oriented numeric codes. Little benefit is seen in the irregular SPECint benchmarks. A loop threading experiment is run on the simulator for the Atlas chip-multiprocessor. The simulator is configured to recognize backwards taken branches. The target of these branches is marked as a thread entrance point. The control flow predictor then attempts to learn and repeat the program control flow as it jumps from loop target to loop target. In the case where the code spends large amounts of time in inner loops, this technique works very well. The simulator achieves excellent results on numeric codes, similar to the results found by others. However, the motivation for adding speculation techniques to a chip multiprocessor is to achieve performance on non-numeric codes. Numeric codes can often be parallelized by the compiler, and this approach is preferred.

Figure 6 shows the results of performing loop threading on SPECint95. The graph shows speedup on the y-axis (8 processors are used in all experiments). The bars

show the distribution of cycles for each benchmark. The black section is achieved speedup (perl achieves a speedup of 1.5, for example). The other shaded regions show where lost cycles are going, divided into five regions:

- Losses due to control mispredicts (white region)
- Losses due to load balance and coverage (lightly shaded region)
- Losses due to data value mispredictions (darkly shaded region)
- Losses due to non-speculative data cache misses and speculative write buffer full stalls (vertically graded)
- Losses due to thread startup and retirement overhead (horizontally graded)

The data in this and subsequent similar graphs is calculated as follows. First, the simulator is set for perfect control prediction, perfect value prediction, perfect caches, and zero overhead. The only source of performance loss in this case is load balance and coverage. The difference between peak speedup (8), and the results of ideal simulation give the fraction of execution bandwidth lost to load balance and coverage. Similar simulations are run with the various penalties selectively enabled, allowing for a measurement of execution bandwidth lost. Finally, a simulation is run without any ideal assumptions. This gives the actual performance. All results are presented as speedup, calculated as the execution time of 8 processors over the execution time of 1 processor. Speedup is shown, rather than IPC, because this research assumes a single-chip multiprocessor and then addresses techniques to improve performance on non-numeric programs. There is a strong belief that a multiprocessor is a solid candidate for future microprocessors, even without the techniques presented here [7][16].

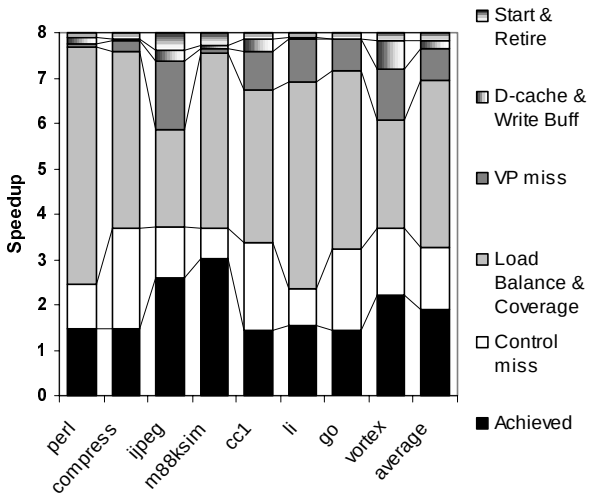

Figure 6: Loop threads

Loop threads do not perform well in SPECint95. It is notable that the results presented here are better than those achieved by others doing loop threads on these benchmarks. This is due to the more aggressive correlated value predictor, and the fine-grained recovery mechanisms of the Atlas processor. From these results, it is

clear that load balance and coverage account for the majority of lost cycles. This should be expected because the SPECint applications are not loop intensive. The exceptions are ijpeg, m88ksim, and (to a lesser extent) vortex, which display some looping behavior. Even in ijpeg, however, many important loops iterate only 8 times and contain restrictive hard-to-predict data dependencies.

Table 2 summarizes simulator results for the loop threading experiments. The 'avg. size' is the average size in dynamic instructions of all the threads in the benchmark. 'Done Speculatively' is the fraction of work performed on speculative processors. Control and data miss rates are the average misprediction rates over all threads in the benchmark.

Benchmark	Speedup	Avg. Size	Done Speculatively	Control Miss rate	Data Miss rate
perl	1.49	37	0.374	0.065	0.306
compress	1.47	109	0.392	0.258	0.267
ijpeg	2.59	61	0.672	0.101	0.247
m88ksim	3.01	49	0.718	0.055	0.066
cc1	1.43	124	0.332	0.245	0.282
li	1.54	101	0.383	0.143	0.253
go	1.44	86	0.346	0.268	0.326
vortex	2.21	198	0.489	0.058	0.128
average	1.90	96	0.463	0.149	0.234

Table 2: Loop threads summary

5.2 Procedure threads

Another approach to dynamic thread partitioning is to target procedure calls. The idea is to execute procedure calls in parallel with the code that follows the procedure. This offers a nice solution to control predictability, because very often the procedure will return to the calling site. In addition, data flow is often favorable around procedures. Many procedures return only an error condition that is easily predictable. The Dynamic MultiThreaded processor (DMT) achieves most of its performance gain from dynamically forking procedure threads [2]. The Atlas simulator was modified to automatically fork the code following a procedure call and execute it speculatively in parallel with the procedure.

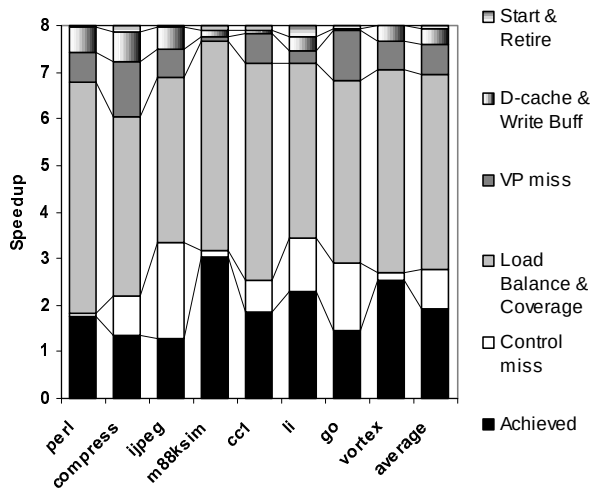

Figure 7: Procedure threads

As can be seen from the results in Figure 7, the main drawback of this approach is again load balance and coverage. Calling patterns vary widely between programs. For example, if a program spends most of its execution time in a large loop without calling procedures, no benefit of procedure threads will be seen. The fact that procedures return to the calling site is also no guarantee of perfect control prediction. To achieve perfect control flow prediction, the machine must be able to speculate control eight levels deep. In the case of procedure threads, this means guessing the next eight procedures to be called.

An important aspect of procedure threads is their ability to expose very large, control predictable threads. The average thread size in SPECint95 is over 570 instructions, with a control mispredict rate of only 11%. This aspect of procedure threads can be very useful in architectures that benefit from deep lookahead execution. For example, deep thread lookahead could be used in an aggressive data prefetching implementation.

Large threads can increase the chances of a data value mispredict, often meaning that large amounts of work will be discarded. On average, however, this approach is more profitable than loop threading for SPECint95. Table 3 summarizes simulator data on procedure threads.

Benchmark	Speedup	Avg. Size	Done Speculatively	Control Miss rate	Data Miss rate
perl	1.75	475	0.503	0.028	0.155
compress	1.36	549	0.362	0.221	0.329
ijpeg	1.27	981	0.264	0.023	0.260
m88ksim	3.05	1305	0.698	0.013	0.025
cc1	1.86	167	0.513	0.150	0.283
li	2.28	532	0.634	0.150	0.155
go	1.44	280	0.337	0.255	0.347
Vortex	2.52	276	0.617	0.038	0.142
Average	1.94	571	0.491	0.110	0.212

Table 3: Procedure threads summary

5.3 Fixed interval threads

The thread policies considered so far suffer severe problems due to load balance and coverage. The next threading technique considered is an idealization of that used by the trace processor [18]. The idea is to use fixed instruction length intervals of the dynamic instruction stream. The trace processor partitions threads on trace cache line boundaries. In an ideal trace cache, each line is full with valid instructions. However, this is not possible due to microarchitectural restrictions, such as a maximum number of branches per line. An ideal fixed interval partitioning policy achieves a perfect load balance, but makes no attempt to address data flow or control flow issues.

Figure 8 shows results of simulating fixed length (16 instruction) threads on the Atlas multiprocessor. Simulations were also run with longer and shorter length threads. Only the 16 instruction length thread results are presented, because this length achieves the best performance. The tradeoffs involved in thread sizing will be discussed in Section 6.2. As anticipated, this approach does a much better job with load balance – nothing was lost to load balance and coverage in all cases. Notice that the fraction of peak speedup lost to startup/retirement is roughly the same for every benchmark. This is the amount lost due to the overhead of processing the relatively short threads. Every processor has a minimum four cycle pipeline startup latency that is the principle contributor here, as very small threads are not able to amortize this cost. Slight variations in the startup/retirement amount between benchmarks are due to the different number of value predictions being updated, and the amount of state that needs to be committed to the L2 cache at thread retirement.

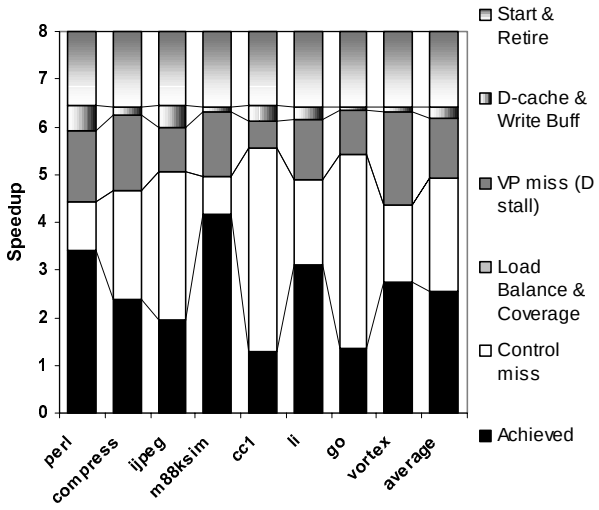

Figure 8: Fixed interval partitioning (16 instruction)

Fixed interval threads stand to gain from better value prediction. This is important because value predictors will improve as researchers develop a deeper understanding of the phenomena. Overall, the fixed interval policy provides better performance than loop or procedure threads. Fractions of the execution bandwidth are lost due to all

penalty aspects, but, a significant portion is not lost to any one area. Table 4 summarizes data for fixed interval threads.

Benchmark	Speedup	Avg. Size	Done Speculatively	Control Miss rate	Data Miss rate
perl	3.41	16	0.824	0.050	0.251
compress	2.38	16	0.713	0.122	0.331
ijpeg	1.96	16	0.648	0.185	0.364
m88ksim	4.17	16	0.835	0.031	0.070
cc1	1.30	16	0.391	0.274	0.396
li	3.12	16	0.811	0.085	0.342
go	1.36	16	0.482	0.346	0.477
vortex	2.74	16	0.736	0.062	0.281
average	2.55	16	0.680	0.144	0.314

Table 4: Fixed interval thread summary

6 MEM-slicing algorithm

A new algorithm has been developed that improves upon current dynamic partitioning algorithms. The algorithm works as follows. Upon starting a thread, the processor first skips a fixed number of dynamic instructions, designated the *skip distance*. This insures that all threads are at least a minimum length, avoiding very small threads. Next, dynamic execution continues while the processor searches for an appropriate place to begin the next thread. The instruction chosen is called the *slice instruction*. Upon finding a suitable instruction, execution stops at the instruction just before the slice instruction. If a suitable slice instruction is not found, the thread is cut off at some *maximum length.*

This process is illustrated in Figure 9. The three defining variables of the approach are the skip distance, the slice instruction, and the maximum thread length. Finding good solutions for these three variables is the subject of following sections. The goal is to find load balanced, data and control predictable threads by tuning these three parameters.

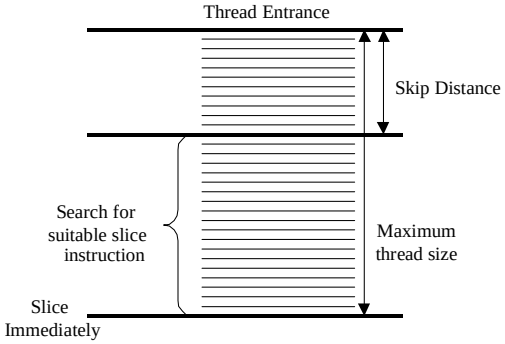

Figure 9: Skip-search-slice algorithm

6.1 Looking for predictable instructions

In [4], a profiler was presented that collected data on the relative merits of every static instruction as a thread boundary. A large variability was found between different static instructions. Some make very good threads, while others make very poor threads. A central idea that emerged from the work is that certain instruction types are better than others for starting a thread boundary, and should be favored as the slice instruction.

Numerous experiments were conducted to determine the best instruction type to use as a slice instruction. Profiler data suggested that backwards branches and stores are the best for data predictability. However, section 5.1 showed that loops (as indicated by backward branches) do not make good threads because of load balance. Using backward branches and stores as the slice instruction leads to similar load balance problems. It was determined from experimentation that using memory instructions (all loads and stores) works well. Memory instructions are very common in integer programs. After the minimum skip distance has been executed, it is typically not long until a memory instruction is encountered. This has the nice effect of making all threads relatively the same size. Profiler data also indicated that memory instructions offer better thread data predictability than arithmetic or control instructions.

A possible explanation why memory instructions offer better thread data flow is that it is a side-effect of register allocation. In register allocation, the compiler strives to put data with short reuse distances into registers, and longer-lived variables into memory. It is desirable to have communication locality within the thread, and longer-lived variables spanning threads. This is exactly the behavior that occurs when memory instructions begin and break threads.

6.2 Size analysis

The previous section determined that memory instructions should be used as the slice instruction. The remaining variables in the partitioning algorithm are the skip distance and the maximum distance. When using memory instructions as the slice instruction, the maximum distance is not important, as a memory instruction is usually encountered soon after the minimum skip distance. The remaining interesting design variable is determining the skip distance.

The MEM-slice algorithm was implemented in the Atlas chip multiprocessor simulator, and the skip distance was varied from 4 to 64. Results are seen in Figure 10. The characteristic shape is that performance climbs as threads go from very small to around 16 instructions, and then performance drops off with larger threads.

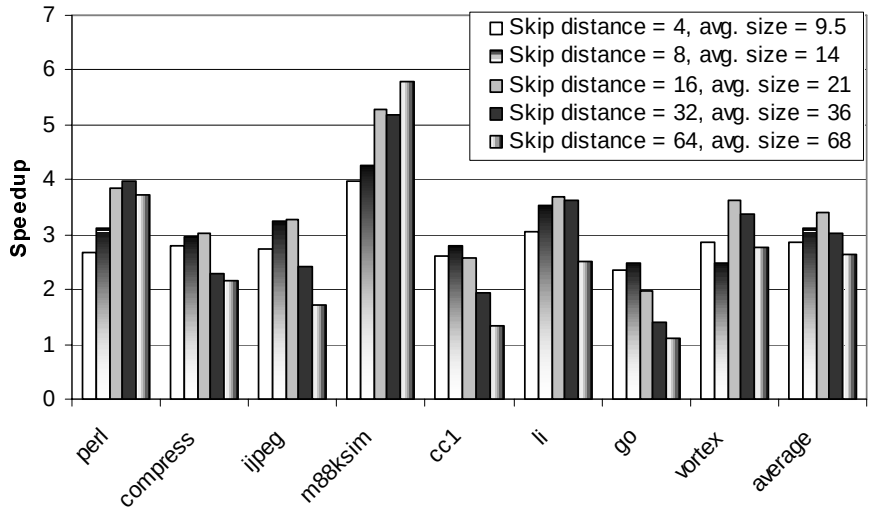

Figure 10: Skip distance

Figure 11 examines the cause of this behavior. Very small threads are dominated by load balance and overhead problems. As threads become bigger, load balance and overhead issues become less important. Control flow is just the opposite. With small threads, it is easier to make control predictions than with large threads. As shown in the figure, these two forces oppose each other. Optimal performance comes in the middle (the apex of the "achieved" triangle).

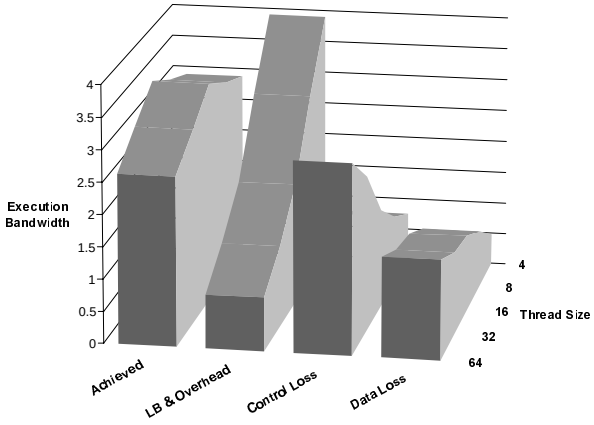

Figure 11: Size Analysis

6.3 MEM Slicing Results

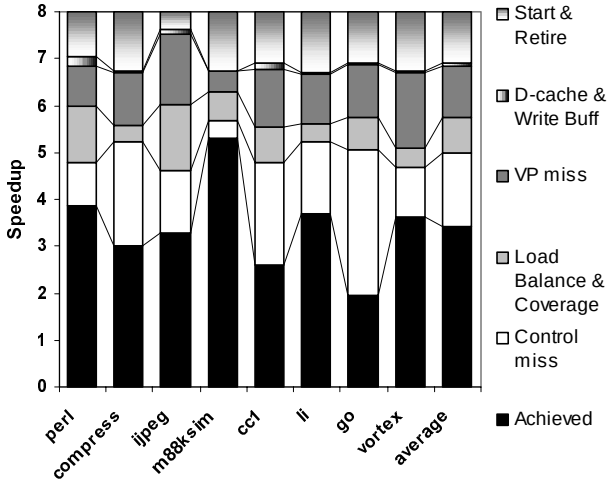

Figure 12: MEM-sliced threads

Figure 12 and Table 5 show results of the MEM-slice algorithm with a skip distance of 16. The next section compares all previous dynamic partitioning policies and the new MEM-slice policy.

Benchmark	Speedup	Avg. Size	Done Speculatively	Control Miss rate	Data Miss rate
perl	3.85	21	0.836	0.044	0.186
compress	3.02	19	0.766	0.108	0.170
ijpeg	3.27	33	0.776	0.076	0.257
m88ksim	5.29	20	0.869	0.020	0.029
cc1	2.59	19	0.732	0.124	0.287
li	3.68	18	0.828	0.069	0.197
go	1.96	19	0.623	0.225	0.339
vortex	3.64	18	0.798	0.035	0.141
average	3.41	21	0.779	0.088	0.201

Table 5: MEM-slice (skip 16) summary

6.4 Comparative Results

Figure 13 compares dynamic partitioning algorithms, showing data arithmetically averaged over all benchmarks. The surprising result is that the primary benefit of the MEM-slicing algorithm is a significant reduction in control mispredicts – not data mispredicts as was expected. Data flow and control flow are intricately linked in may complex ways. Results suggest that slicing at memory operations provides better thread data flow. It is postulated that this result is a byproduct of complex analysis already being performed by the compiler's register allocator. Given that memory operations provide better thread data flow, it is reasonable that memory operations should also provide better thread control flow.

In the Atlas architecture, control predictability is more important than data predictability. If a thread is control mispredicted, all speculative work after the misprediction is discarded. When a value is mispredicted, the thread provides fine-grained recovery so only affected instructions need reexecute. Other, more deeply speculated threads are not squashed. Often, more future threads complete correctly in the presence of a less speculative data value misprediction.

Interestingly, the analysis in [4] intended to find solutions for data flow and ended up with a solution for control flow.

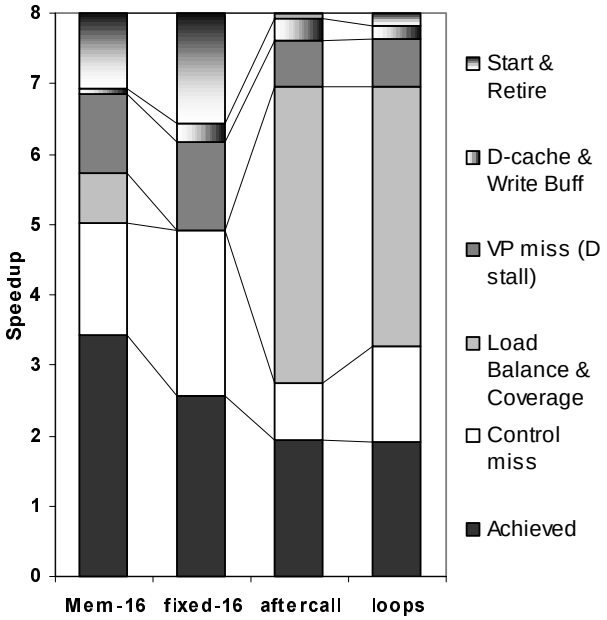

Figure 13: Comparison of dynamic partitioning algorithms

The important lesson learned is that the weakest aspect of a partitioning algorithm limits performance. For example, procedure threads have excellent control and data predictability. But load balance is poor, and as a results overall performance is poor.

The experiments also suggest that load balance and control flow are more important than data flow, especially with fine-grained data value misspeculation recovery. Of the current dynamic partitioning algorithms studied, fixed intervals perform the best. Load balance is completely solved, while data and control flow can be managed by keeping the threads small. The MEM-slice algorithm improves on fixed interval partitioning by improving control predictability.

7 Summary and Conclusions

Speculative multithreading is emerging as a promising technique to extract parallelism from non-numeric applications. A critical problem that must be addressed is how to partition the program into threads. Partitioning dynamically allows for parallel execution of "dusty-deck" binaries.

This paper provides a quantitative comparison of existing dynamic partitioning algorithms in the context of a fixed microarchitecture. Two current approaches are to partition loops and procedures. While these approaches both provide good control and data predictability, they suffer severe load balance and coverage problems. The integer benchmarks are not loop oriented, and calling patterns and procedure sizes vary significantly between and within benchmarks. Over half the execution bandwidth is lost to load balance and coverage when doing procedure and loop threading.

Fixed interval partitioning eliminates load balance and coverage problems, as all threads are exactly the same size over the entire program. However, this approach suffers greater control and data penalties.

A new dynamic partitioning algorithm, called MEM-slicing, is introduced. This technique improves on fixed interval partitioning by being more selective about thread boundaries. By slicing threads only on memory operations, threads can be made larger and more predictable than fixed interval threads. This comes at a cost of load balance, but overall the performance of MEM-slicing greatly exceeds other dynamic partitioning algorithms (34-80% improvement).

A fundamental tradeoff exists between overhead and predictability. As threads are made smaller, they become more predictable. This comes at the cost of increased overhead. Large threads reduce overhead costs, but increase penalties associated with data and control mispredicts. An optimal size exists somewhere between large and small threads. For the simulated Atlas chip-multiprocessor, 16-32 instruction long threads provide the best performance.

It is possible to expect good parallel performance from unmodified sequential binaries using aggressive speculation techniques. Current results suggest speedups of around 3 on 8 processors. These results will improve as partitioning and speculation techniques mature.

Acknowledgments

This work is supported by the DARPA Low Power Electronics (LPE) and Advanced Microelectronics (AME) Programs as well as the Semiconductor Research Corporation.

References

[1] D. Burger, T.A. Austin, "The SimpleScalar Tool Set, Version 2.0", Technical Report #1342, University of Wisconsin-Madison Computer Science, June 1997.

[2] H. Akkary, "Dynamic MultiThreaded Processor," Ph.D. thesis, Portland State University, June 1998.

[3] H. Akkary and M. Driscoll, "A Dynamic Multithreaded Processor," in Micro-31, Dec. 1998

[4] L. Codrescu and S. Wills, “Profiling for Input Predictable Threads,” in *ICCD-98*, pp. 558-565, October 1998.

[5] L. Codrescu and S. Wills, “Architecture of the Atlas Chip-Multiprocessor: Dynamically Parallelizing Irregular Applications”, in ICCD-99

[6] L. Codrescu and S. Wills, “The HLG correlated value predictor,” Pica Group Technical Report 10-98-A, in submission. Available from http://www.ece.gatech.edu/users/lucian

[7] B. Dally and S. Lacy, "VLSI architecture: past, present, and future", *in 20th ARVLSI*, March 1999.

[8] M. Dorojevets and V. Oklobdzija, “Multithreaded Decoupled Architecture,” Int. J. High Speed Computing, 7(3), pp. 465-480, 1995.

[9] Pradeep K. Dubey, Kevin O’Brien, and Charles Barton. “Single-program speculative multithreading (SPSM) architecture: Compiler-assisted fine-grained multithreading,” in *PACT,* pp. 109-121, June 1995

[10] M. Franklin, “The Multiscalar Architecture” Ph.D thesis, University of Wisconsin – Madison, 1993

[11] S. Gopal, T.N. Vijaykumar, J. Smith, and G. Sohi, “Speculative Versioning Cache,” in *HPCA-4*, February 1998.

[12] Venkata Krishnan & Josep Torellas, "Executing sequential binaries on a multithreaded architecture with speculation support", *in HPCA-4*, February 1998

[13] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen, “Value locality and data speculation,” in *ASPLOS-7,* pp. 138-147, October 1996

[14] P. Marcuello, A. Gonzalez, and J. Tubella, "Speculative Multithreaded Processors,”,in *12th Int’l Conference on Supercomputing*, 1998.

[15] Andreas I. Moshovos, Scott E. Breach, T.N. Vijaykumar, Gurindar S. Sohi, "Dynamic Speculation and Synchronization of Data Dependences", in *ISCA-24*, June 1997

[16] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chung, “The Case for a Single-Chip Multiprocessor,” In *ASPLOS-7*, October 1996.

[17] Jeffery Oplinger, David Heine, Shih-Wei Liao, Basem A. Nayfeh, Monica S. Lam, and Kunle Olukotun, “Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor,” Tech. Rep. CSL-TR-97-715, Computer Systems Laboratory, Stanford University, May 1997

[18] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith, et al, “Trace Processors”, in *Micro-30,* pp. 68-74, December 1997

[19] V. Sarkar and J. Hennessy, "Partitioning parallel programs for macro-dataflow", In *Conference Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pp. 192-201, 1986.

[20] Gurindar S. Sohi, Scott E. Breach, and T.N. Vijaykumar, “Multiscalar processors,” *in ISCA-22*, pp. 414-425, June 1995

[21] J. Gregory Steffan and Todd C. Mowry, “The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization,” *in HPCA-4*, February 1998

[22] J.-Y. Tsai and P.-C. Yew, “The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation”, *in PACT*, October 1996

[23] J. Tubella and A. Gonzalez, "Control Speculation in Multithreaded Processors through Dynamic Loop Detection," in *HPCA-4*, February 1998.

[24] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *ISCA-22,* pp. 392-403, June 1995

[25] T.N. Vijaykumar, "Compiling for the Multiscalar Architecture", Ph.D. Thesis, U. Wisconsin - Madison, Jan 1998.

[26] T.N. Vijaykumar and Gurindar S. Sohi, "Task Selection for a Multiscalar Processor", in Micro-31 Dec 1998.

[27] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, , "Baring It All to Software: RAW Machines", IEEE Computer, pp. 86-93, September 1997.