Appears in IASTED International Conference on Parallel and Distributed Systems (Euro-PDS), July 1-3, 1998, Vienna, Austria.

SIMULATION STUDY OF MULTITHREADED VIRTUAL PROCESSOR

BEN LEE, HANTAK KWAK, and RYAN CARLSON

Department of Electrical and Computer Engineering
Oregon State University
Corvallis, OR 97331, USA
{benl, hantak, carlsor}@ece.orst.edu

SUK-HAN YOON and WOO-JONG HAN

Computer Division
Electronics and Telecommunications Research Institute
Taejon, Korea
{shyoon, wjhan}@computer.etri.re.kr

ABSTRACT[1]

This paper proposes the Multithreaded Virtual Processor (MVP) architecture model as a means of integrating the multithreaded programming paradigm and a modern superscalar processor with support for fast context switching and thread scheduling. In order to validate our idea, a simulator was developed using a POSIX compliant Pthreads package and a generic superscalar simulator called SimpleScalar glued together with support for multithreading. The simulator is a powerful workbench that enables us to study how future superscalar design and thread management should be modified to better support multithreading. Our simulation studies show that the performance of MVP is highly sensitive to three interrelated factors: (a) the data set size relative to the cache size, (b) the number of hardware contexts, and (c) the amount of locality that can be exploited among the threads. These results also show that multithreading creates an additional stress on the fetch bandwidth. In summary, our results show that in general the performance improvement is obtained from not only tolerating memory latency, but also from lower cache miss rates due to exploitation of data locality. Therefore, the MVP model is well worth investing extra silicon space to support multithreading. Moreover, as the gap between processor speed and memory latency becomes wider, the need to provide multithreading support will become more crucial.

Key Words: Multithreading, context-switch, thread scheduling, multiple hardware contexts, and threads.

1. INTRODUCTION

Small-scale shared-memory multiprocessors, such as Symmetric Multiprocessors (SMPs), have become the dominant form of parallel machines for commercial as well as scientific computing. This dominance has been due to the availability of powerful, and yet cheap, commodity microprocessors that can issue multiple instructions per cycle and a global shared-memory that significantly eases the programming task. In the past few years, there also have been tremendous efforts in the development of large-scale parallel computers and parallel computing clusters. Another equally important trend is the efforts to provide shared-memory programming on these machines. These machines span the spectrum; from systems with physically distributed memories with hardware support for cache-coherence, to Networks of Workstations (NOWs) interconnected by a LAN or WAN with software support for shared-memory abstraction.

As parallel machines become larger, so does the memory latency. Therefore, the proportion of the processor time actually spent on useful work keeps diminishing. The memory latency problem is attributed to two main bottlenecks: memory system and communication pathways. Studies show that the speed of commercial microprocessors has increased by a factor of twelve over the past ten years while the speed of memories has only doubled [9]. Therefore, memory latency in terms of processor clock cycles has grown by a factor of six in ten years. More importantly, the gap between processor cycle time and memory cycle time will no doubt continue to increase in the future. Multiprocessors and multicomputers greatly exacerbate the memory latency problem. In SMPs, contention due to the shared bus located between the processor's L2 cache and the shared main memory subsystem adds additional delay to the memory latency. The memory latency problem becomes even more severe for scalable Distributed Shared Memory (DSM) systems, since a miss on the local memory requires a request to be issued to the remote memory and a reply to be sent back to the requesting processor. Stalls due to the round-trip communication latency are, and will continue to be, the dominating factor that limits the performance of scalable DSM systems in the future.

Multithreading is a technique that has emerged as one of the most promising and exciting avenues to tolerate the increasing memory latency. A multithreaded system contains multiple "loci of control" (or threads) within a single program; the processor is shared by these multiple threads leading to higher utilization. The processor may switch between threads not only to hide memory latency, but other long latency operations, such as I/O latency. The processor may also interleave instructions from multiple threads on a cycle-by-cycle basis, which minimize

[1] This research was supported in part by the Electronics and Telecommunications Research Institute (ETRI), Taejon, Korea.

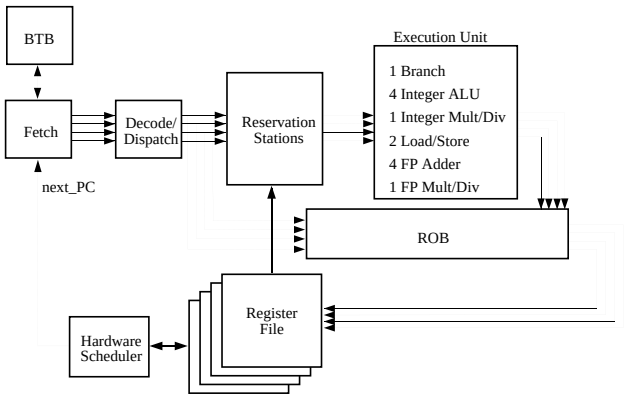

Figure 1: The organization of the MVP.

pipeline breaks due to dependencies among instructions within a single thread.

The idea of multithreading is not new. Fine-grained multithreading was implicit in the dataflow model of computation [10]. Multiple hardware contexts (i.e., register files and PSWs) to speed up switching between threads were implemented in systems such as HEP [16] and Tera [3]. However, these systems require considerable modification to the underlying architecture. There also has been an effort to integrate multithreading support on an existing processor—MIT Alewife machine uses a modified SPARC processor called Sparcle [1]. However, Sparcle is based on an outdated processor design; therefore, it is unclear what effect multithreading will have on modern superscalar architectures. Multithreading has also been extensively used strictly as a programming paradigm (i.e., software-controlled multithreading) on general-purpose hardware to increase applications' throughput and responsiveness and to exploit thread parallelism on SMPs [7]. However, software-controlled multithreading systems, such as Pthreads [6] and Solaris threads [7], lack the hardware features necessary to detect and handle cache misses and therefore the ability to hide memory latency.

In light of the aforementioned discussion, this paper presents the development of the *Multithreaded Virtual Processor* (MVP), which exploits the synergy between the multithreaded programming paradigm and the well-designed wide-issue microprocessors. MVP proves that by providing an adequate hardware support to an existing superscalar core, we can take full advantage of the increasingly popular and powerful programming tools that exploit thread parallelism. Moreover, as the feature size of logic devices reduces, we feel the silicon area could be put to better use by providing support for multithreading.

Our simulation studies show that performance of MVP is greatly influenced by three main factors: (a) the data set size relative to the cache size, (b) the number of hardware contexts supported, and (c) the amount of locality that can be exploited among the threads. These results also show that context switching among threads creates an additional stress on the fetch bandwidth. Our overall results show that in general the performance improvement obtained with MVP is well worth the investment in providing support for multithreading.

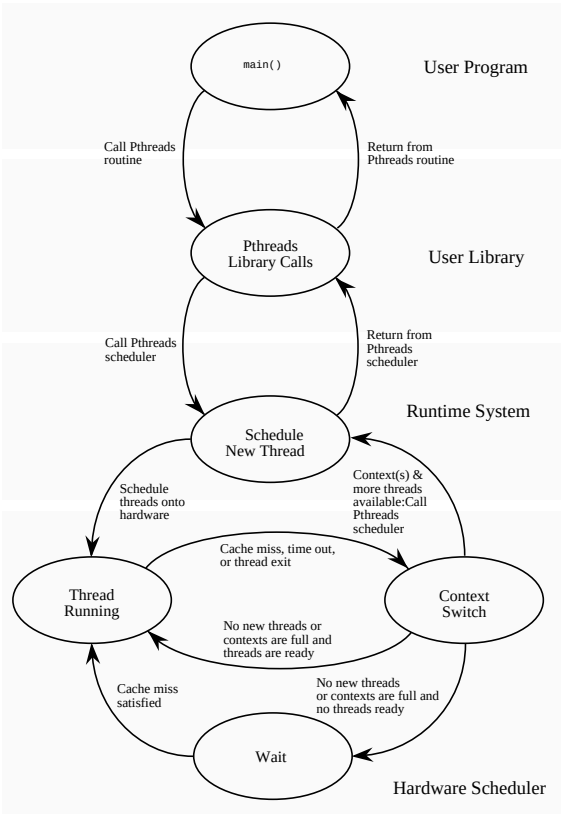

Figure 2: MVP execution model.

2. MVP

The objective of the proposed MVP is to extend the software-controlled multithreading model with hardware support for tolerating memory latency, and yet provide a transparent view to the programmer. The organization of MVP is shown in Figure 1. It consists of a conventional superscalar processor core augmented with the Hardware Scheduler and multiple Register Files (RFs), each RF representing a thread context. The responsibility of the Hardware Scheduler is to maintain the control of thread states that have been scheduled onto MVP.

Threads are created, managed, and scheduled using a POSIX compliant Pthreads package [6]. Pthreads' runtime scheduler schedules user created threads onto the multiple hardware contexts. Once threads are scheduled onto MVP, the Hardware Scheduler context-switches between threads whenever a long latency memory operation is detected by the memory-management unit. When a thread returns, the control is returned to the Pthreads' scheduler. The interaction among the user program, Pthreads function calls, Pthreads scheduler, and the Hardware Scheduler is shown in Figure 2. The functionality of each state is explained below:

- `main()`- MVP is executing the main user program. When a Pthreads routine is invoked, a state transition is made to Pthreads Library Calls.
- Pthreads Library Calls - Executes a Pthreads routine. After executing a Pthreads routine, it will either return to `main()` or call the Pthreads scheduler (Schedule New Thread state).

- Schedule New Thread - Depending on which Pthreads routine called the Pthreads scheduler, it will perform either one of the following operations:
  - Checks to see if a thread to be scheduled from the priority queue (PQ), which is maintained by the Pthreads scheduler, has a higher priority than the currently running thread. If so, the thread in the PQ is scheduled onto a hardware context in MVP; otherwise, returns from the Pthreads scheduler. Also checks and processes any signals.
  - Selects a thread from the PQ and schedules to an available hardware context. Also checks and processes any signals.
- Thread Running - Runs a thread that is in a hardware context. A transition to the Context Switch state occurs when a cache miss, a time out, or a thread exits.
- Context Switch - Depending on the state of the machine, it will perform one of the following operations:
  - Context-switches to one of the ready threads in MVP (i.e., Thread Running state).
  - Calls the Pthreads scheduler if a hardware context is available and threads are waiting to be scheduled in the PQ (i.e., Schedule New Thread state).
  - Waits if hardware contexts are full but no threads are ready, and no threads are waiting to be schedule from the PQ.
- Wait - Waits for a thread to become runnable (waiting for a long latency memory operation). When a thread becomes runnable (i.e., its cache miss has been satisfied), a transition is made to Thread Running state.

There were two design choices made in the process of developing the MVP execution model. First, instead of scheduling all the threads into the hardware contexts at once, the decision was made to schedule them one at a time. The reason for this was because scheduling one thread at a time is closer to how the original Pthreads scheduler accomplished the scheduling and thus less modification to the code was required. We also felt that gang scheduling the threads would not necessary provide improvement in performance because of the minimum overhead of the Pthreads scheduler. Second, when a thread execution returns and thus a hardware context becomes available, the decision was made to immediately schedule a new thread from the PQ rather than context switching to a thread already residing in one of the hardware contexts. This design choice was based on what we learned from our preliminary studies—it is always better to keep hardware contexts occupied with threads [14].

3. SIMULATION RESULTS

In order to study viability of the MVP model, a detailed simulator was developed by integrating Pthreads and SimpleScalar [5] with support for multithreading. Simulation studies were conducted using the following assumptions:

- Functional unit latencies were based on Table 1.
- Cache and main memory organizations and their latencies were based on Table 2. We assumed a two-level, i.e., L1 and L2, blocking cache scheme. The main memory latency used in the simulation is rather conservative compared to the current technology, e.g., UltraSparc IIi has a main memory latency of 72 cycles [13]. However, we expect the memory latency to grow in the future. Moreover, for multiprocessor systems, the shared-bus between processors' lower level cache and the main memory adds to the latency [8].

Table 1: Instruction latencies for various FUs.

Functional Unit	Latency	Pipelined
Integer ALU	1	Yes
Load/Store Unit	2	Yes
Integer Multiply	3	Yes
Integer Division	12	No
FP Addition	2	Yes
FP Multiplication	4	Yes
FP Division	12	No

Table 2: Cache and Main Memory latencies.

	L1 I-cache	L1 D-cache	L2
Size	16KBytes	16KBytes	256/512 KBytes
Associativity	DM	4-way SA	4-way SA
Line size	32 Bytes	32 Bytes	32/64 Bytes
Latency (hit)	1	1	6
Latency (miss)	6	6	100

- Context switching to a new thread is initiated when an L2 cache miss is detected. L1 cache misses were not supported since the latency is minimal (6 cycles) and therefore not worth context switching to a new thread. However, a context-switch can be initiated at any level of the memory hierarchy as long as sufficient latency exists.
- The number of instructions fetched, decoded, and dispatched is 4. The number of entries in the Reservation Stations and ROB were each assumed to be 16.
- For branch prediction, we used a 2K-entry Branch Target Buffer (BTB) with 2-bit branch prediction bits.
- The process of switching from one hardware context to another involves (a) simply turning off one register bank and turning on another register bank, (b) flushing the ROB, and (c) fetching from the new context. Assuming this is supported entirely in hardware, this process is very similar to recovering from a miss-predicted branch and requires a penalty of 3 cycles.
- The number of threads created for each simulation run was kept equal to the number of hardware contexts. We found that varying the number of threads created had minimal effect on the overall performance. Adding more threads incrementally increased the amount of software overhead required to schedule the threads onto the hardware contexts. However, as long as threads are not created unnecessarily, the software overhead has minimal effect on the overall performance.

Five benchmark programs were selected to evaluate the performance of MVP. Matrix Multiplication (MMT) and Gaussian Elimination (GE) were hand-coded. The other three benchmarks, MP3D, Radix Sort (RS), and Fast Fourier Transform (FFT) were selected from the SPLASH-2 benchmark suite [18]. The SPLASH-2 benchmarks were originally constructed for shared-memory machines and uses ANL macros to create and manage the threads. To port the SPLASH-2 benchmarks to the simulator, the ANL macros were replaced with their Pthreads equivalents. No attempts were made to optimize the codes and no special hardware synchronization mechanism was provided, as was done in [11].

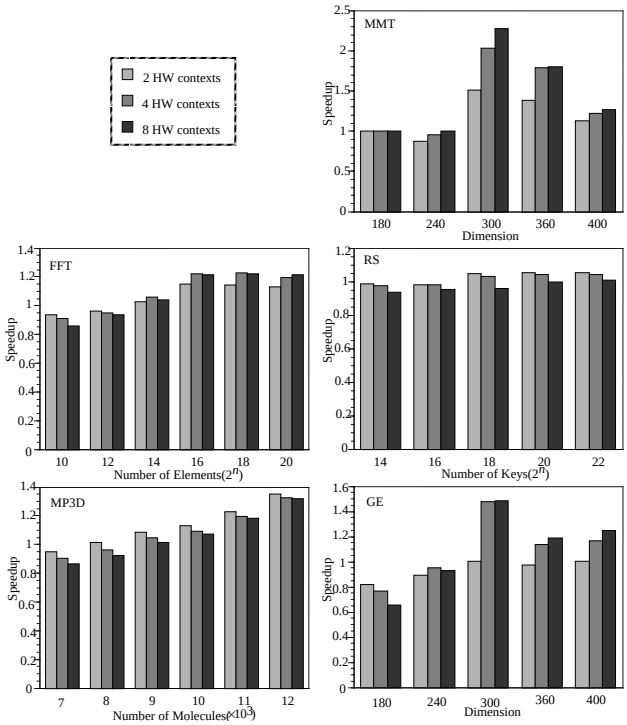

Figure 3: Speedup results for various benchmarks.

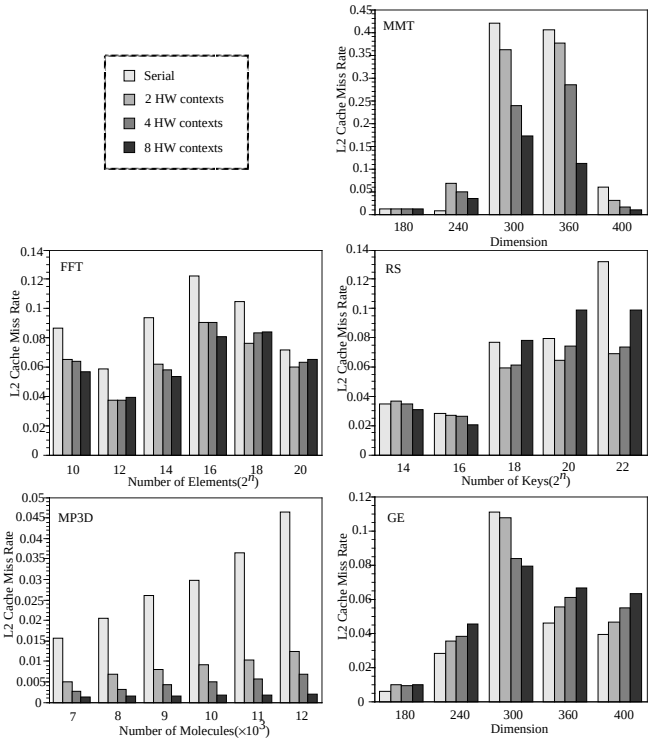

Figure 4: L2 cache miss rates for various benchmarks.

Four sets of simulation runs were performed for comparison purposes. The first set was obtained by running a single-thread version of the benchmarks on SimpleScalar (Serial version). The other three sets were obtained by running on MVP with 2, 4, and 8 hardware contexts. These results were obtained by simulating approximately 160 million instructions (MP3D) to 1.1 billion instructions (GE and RS).

Figure 3 shows the relative performance of MVP for the benchmarks. The results were normalized relative to the performance of the serial versions. Figure 3 shows that as the data sets become large, MVP begins to overcome its overhead and performs better than the serial cases. An example of this effect is displayed by MP3D. FFT also shows great performance improvement as data size increases as the algorithm begins to take advantage of latency tolerance of multithreading. Another interesting effect is that the use of more hardware contexts does not necessarily result in improved performance, as seen in both RS and MP3D. This effect is the result of the benchmarks' high synchronization requirements and small parallel portions. Although the performance degrades as the number of contexts increases, the performance margin narrows as the problem size increases. In essence, the 4 and 8 hardware contexts would eventually outperform the 2-hardware contexts case. Both GE and MMT resulted in improved, but varied performance. In GE, the performance of MVP for the 300 case appears to be based on a lower L2 miss rate (compared to the serial version) caused by the good mapping of the 300 by 300 matrix into the L2 cache. MMT performance also appears to be seriously affected by cache effects as the 300 case resulted in a lower L2 miss rate (compared to the serial version), while the 240 case resulted in a much higher L2 miss rate.

Another interest is the effect that multithreading produces on the caches. In order to gain a good understanding of how the L2 cache is affected, two sets of graphs were obtained. The first set, monitored the L2 miss rate to determine the effects that are seen by the L2 to main memory bus. The second set, monitored the number of accesses received by the L2 cache. In essence, this data set monitors the effect on the L1 cache by the simulated benchmarks.

The data graphs shown in Figure 4 give some rather interesting results. It is apparent that, for the most part, the L2 miss rates for MVP are lower than the serial versions. This effect is seen in all the benchmarks except GE. Lower miss rates are due to the fact that the data sets used by these programs have a very high locality. This locality creates a situation where one thread can help another. In essence, one thread can cause a cache miss that will bring in the data that another thread will need later. To illustrate this point, consider RS. When one of its threads accesses the global histogram for a data value, it also brings in other data values that it does not need, but other threads do. Thus, another thread will find its global histogram data already in the cache and will not generate a cache miss. The serial programs cannot do this and thus result in worse performance by generating a cache miss for a line, using the small portion of data off that line, and then replacing the line before the rest of the data on the line has a chance to be used. MMT generates a similar effect by the nature of its algorithm. MMT multiplies a row of one matrix with a column of another. By having the rows of both matrices distributed in a row-wise block manner among threads, when one thread cache misses on a column value, it will load in data that will be used by another thread. This extra data is obtained by the fact that

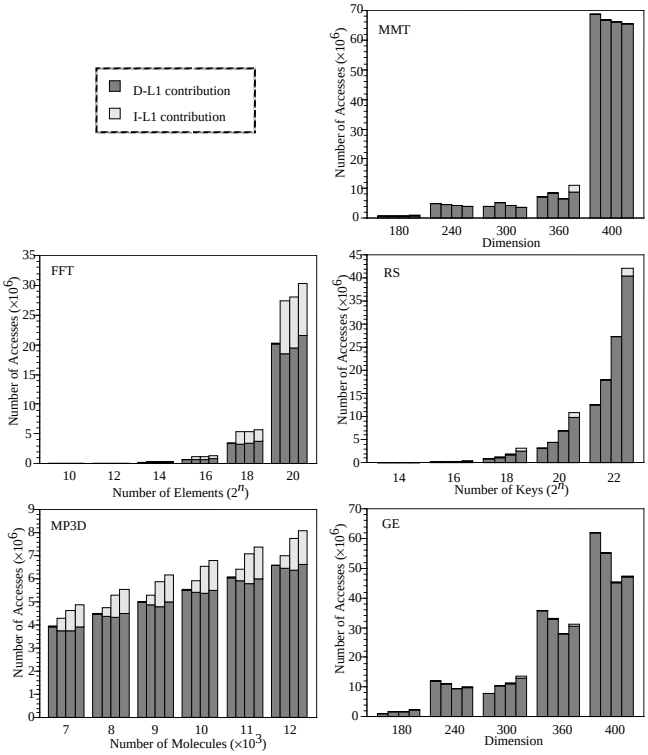

Figure 5: L1 cache miss accesses for various benchmarks. Four graphs for each point represent, from left to right, serial, 2 HW contexts, 4 HW contexts, and 8 HW contexts.

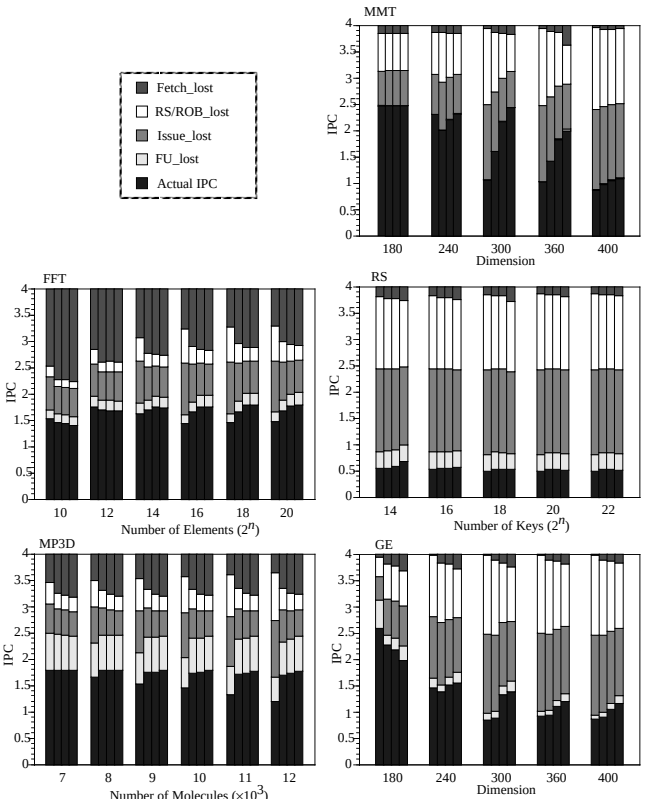

Figure 6: IPC for various benchmarks. Four graphs for each point represent, from left to right, serial, 2 HW contexts, 4 HW contexts, and 8 HW contexts.

a cache line is essentially part of a row in the matrix. When a thread generates a cache miss while looking for a column value, other values belonging to the same row are retrieved at the same time. Therefore, when another thread looks for its column value, the column could already be in the cache. Again, the serial programs tend to replace the data before it can be reused, thus generating more cache misses.

Another effect seen by RS is the increased L2 miss rate as the number of hardware contexts is increased. This is caused by the sorting portion of the algorithm. When the threads sort their individual keys of the array, the data set becomes very disjoint between threads, and the L2 cache miss rate increases. The sorting portion, as seen by the graphs, has a more profound effect as the number of hardware contexts increases. Similarly, GE also uses very distinct and low locality data sets among threads. Therefore, a thread, upon generating a cache miss, would simply bring in more of the rows belonging to the same thread. The result is that the threads compete for space within the L2 and results in a higher L2 miss rate than the serial version.

There also exist two unique effects seen by both GE (at 300) and MMT (at 240), where the graphs see a reversal of the general trend between the miss rates. It is hypothesized that these results are obtained from the simple fact that some data set sizes tend to fit the L2 cache layout much better or much worse than others. Another effect seen in Figure 4 is that RS and FFT exhibit minimum cache miss rates for cases 16 and 12, respectively. At these points, the caches are at the stage where they are nearly full (so compulsory misses are offset and conflict misses are at a minimum). Increasing L2 accesses (i.e., increasing the data set size) would cause an increase in conflict and capacity misses, while decreasing L2 accesses (i.e., decreasing the data set size) would emphasize the effects of the compulsory misses.

Figure 5 shows the effect on L1 caches. For FFT, MP3D, and MMT, the results suggest that while multiple threads work well in the L2 cache, the exploitation of locality is hindered in the L1 cache. The result is that the threads conflict with one-another causing more access to the L2 cache to retrieve the replaced lines. It also appears that the data separation seen with GE and MMT results in an almost opposite effect. With these two benchmarks, the non-local data serves to map the threads better into the L1 caches than the serial versions and the L2 cache sees fewer accesses. However, it is important to note that again, matrix size plays a very important role as both GE (at 300) and MMT (at 360) show completely opposite results compared to the rest of the data gathered for those two benchmarks.

Figure 6 shows the average Instructions executed Per Cycle (IPC). These were collected in order to study any new bottlenecks that multithreading might cause to a superscalar pipeline. IPCs were obtained by dividing the total number of instructions executed by the total number of cycles. The portions of IPC lost were also analyzed from each of Fetch stage, Dispatch stage, and Issue stage. The graphs display the IPC that was lost from the ideal IPC. In other words, the lost IPC that is shown is simply the ideal IPC minus the actual IPC. The graphs are fur-

ther broken down to show what percentage of the total lost IPC was incurred at each of the stages. The pipeline stage bottlenecks that were modeled are: IPC lost due to fetch bandwidth (Fetch_lost), RS/ROB full (RS/ROB_lost), execution units busy (FU_lost), and issue bandwidth limitations (Issue_lost). Decode and Commit stage bandwidths were also observed, but was dropped when it was apparent that bandwidths were never reached in any of the simulations executed.

Looking at the graphs it is clear that multithreading creates an additional stress on the fetch bandwidth. This is due to the fact that program locality is reduced by context switching among threads. Therefore, the fetch bandwidth will have to be improved in order to obtain better performance [8, 15]. Another effect that can be observed is a decrease in IPC lost in the issue stage. Long latency data dependencies are avoided by switching threads on a cache miss. Consequently, more instructions are available to be issued. Also seen by the graphs is almost no change by RS and only small amounts of difference experienced by FFT and MP3D. The reason is the level of synchronization and therefore parallelism experienced by the data sets. RS has much synchronization while FFT and MP3D have some and MMT has none. GE exhibits a similar effect as MMT even though it also has high amounts of synchronization. This effect is because GE has a very large parallel portion in comparison to the synchronized serial portions.

4. CONCLUSION

This paper discussed the simulation study of MVP. Our results show that the performance improvement comes from both tolerating memory latency and exploiting data locality. We are also very encouraged by the potential of the simulator as a workbench for studying the architectural requirements of future superscalar microprocessors. Therefore, we plan to pursue a number of research directions that would further improve the performance of MVP.

The first task is to continually improve the base MVP model in three related directions. First, we plan to study the thread scheduling as a means of improving data locality. Second, we plan to alleviate the bottleneck in the fetch stage. The current research trend is to combine branch prediction with instruction alignment to improve the instruction fetch bandwidth [8]. We plan to pursue this possibility in the context of multithreading. Finally, we plan to provide hardware support for synchronization. The current implementation of MVP relies on software-controlled mutexes and barriers for synchronization. Our experiments indicate that the impact of synchronization overhead is minimal for programs that have relatively large amounts of parallelism between synchronization points, or when the number of synchronizations is small. However, hardware support for synchronization can greatly benefit programs with heavy synchronization requirements.

We are also working on a new architectural model called Dynamic MVP (DMVP). DMVP extends Simultaneous Multithreading (SMT) [11, 12] with dynamic thread generation and speculative thread execution. DMVP has tremendous potential for exploiting both ILP and Thread Level Parallelism (TLP) in programs, and represents an alternative to contemporary wide-issue superscalar processors.

5. REFERENCES

[1] Agarwal, A. *et al.*, "Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors," *Proc. of Workshop on Scalable Shared Memory Multiprocessor*, Kluwer Academic Publishers, 1991.

[2] Agarwal, A., "Performance Tradeoffs in Multithreaded Processors," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 5, Sept. 1992, pp. 525-539.

[3] Alverson, R. *et al.*, "The Tera Computer System," *Proc. Int'l. Conference on Supercomputing*, June 1990.

[4] Ang, B. S., Chiou, D., Rudolph, L., and Arvind, "Message Passing Support in StarT-Voyager," MIT Laboratory for Computer Science, CSG Memo 387, July 1996.

[5] Burger, D. C., Austin, T. M., and Bennett, S., "Evaluating Future Microprocessors—The SimpleScalar Tool Set," UW Computer Sciences Technical Report #1308, July, 1996.

[6] Butenhof, D. R., *Programming with POSIX Threads*, Addison Wesley, 1997.

[7] Catanzaro, B., *Multiprocessor System Architectures*, Prentice Hall, 1994.

[8] Conte, T. *et al.*, "Optimization of Instruction Fetch Mechanisms for High Issue Rates," *Proc. of the 22nd International Symposium on Computer Architecture*, June 1995.

[9] Culler, D. and Singh, J. P., *Parallel Computer Architecture: A Hardware/Software Approach*, Draft, Morgan Kaufmann, 1997.

[10] Lee, B. and Hurson, A. R., "Dataflow Architectures and Multithreading," *IEEE Computer*, Vol. 27, No. 8, 1994, pp. 27-39.

[11] Lo, J. L. *et al.*, "Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading," *ACM Transactions on Computer Systems*, August 1997, pp. 322-354.

[12] Loikkanen, M. and Bagherzadeh, N., "A Fine-Grain Multithreading Superscalar Architecture," *Parallel Architectures and Compilation Techniques* '96, October 1996.

[13] Normoyle, K., "UltraSPARC IIi™ - A Highly Integrated 300 MHz 64-bit SPARC V9 CPU," *HOT Chips IX*, August 1997.

[14] Ortiz, D., Lee, B., Yoon, S. H., and Lim, K. W., "A Preliminary Performance Study of Architectural Support for Multithreading," *30th Hawaii International Conference in System Science*, Software Track, January 7-10, 1997.

[15] Rotenberg, E. S., Jacobson, Q., Sazeides, Y., and Smith, J., "Trace Processor," *Micro-30*, Dec. 1997.

[16] Smith, B., "The Architecture of HEP," in *Parallel MIMD Computation: HEP Supercomputer and applications*, edited by J. S. Kowalik, MIT Press 1985.

[17] Smith, J. E. and Sohi, G. S., "The Microarchitecture of Superscalar Processors," *Proc. of the IEEE*, December 1995.

[18] Woo, S. C. *et al.*, "The SPLASH-2 programs: Characterization and Methodological Consideration," *Proc. 22nd Annual Symposium on Computer Architecture*, June 1995, pp. 24-36.