A Non-Blocking Multithreaded Architecture

Krishna M. Kavi †
David Levine ‡
and
Ali R. Hurson ¥
† The University of Alabama in Huntsville, ‡ The University of Texas at Arlington,
¥ Penn State University

Abstract

In this paper we present a new approach to building multithreaded uni-processors. Our innovativeness stems from an architecture with non-blocking threads where all memory accesses are decoupled from the thread execution. Data is pre-loaded into the thread context (registers), and all results are "post-stored" after the completion of the thread execution. The decoupling of memory accesses from thread execution requires a separate unit to perform the necessary pre-loads and post-stores, and controlling the allocation of hardware thread contexts to enabled threads. This separation facilitates for achieving high locality and minimizing the impact of distribution and hierarchy in large memory systems. We present our preliminary results obtained from a Monte Carlo simulator that compares the performance of the proposed system with conventional architectures for randomly generated threads.

Key Words. Multithreaded architectures, Dataflow architectures, Simultaneous multithreading, Superscalar architectures.

1. Introduction

The performance gap between processors and memory has widened in the past few years and the trend appears to continue in the foreseeable future. This gap can be masked to some extent by multiple levels of cache memories. However, studies have shown that even this does not eliminate processor idle cycles resulting from cache misses and the concomitant cycles to fetch the missing data across the memory hierarchy. Distributed shared memories only compound this problem.

Multithreading has been touted as the solution to minimize the loss of CPU cycles by executing several instruction streams simultaneously. The advent of superscalar architectures make the advantages of the multithreading even more significant. In conventional superscalar systems, instructions from a single program stream are selected for issue, which leads to stalls on data and control dependencies among the instructions. In multithreading instructions from different streams can be selected for issue, thus eliminating pipeline stalls

In our ongoing research, we have bee investigating architectural innovations for improving the performance of multithreaded dataflow systems [Kavi 95, 97]. We have been exploring ways to use instruction, data and I-structure cache memories [Kavi 95, 97], improving performance of such caches by re-using operand memory [Kavi 96], in-place updates of I-structure caches without changing I-structure semantics [Fang 97], and control-flow based scheduling of instructions within a dataflow thread [Fang 97]. This paper describes a new architecture that brings non-blocking (often fine-grained) data driven thread model to conventional load/store microprocessor designs, for the purpose of tolerating memory latencies. In this paper we present our preliminary results.

We will introduce the new architecture in Section 2. A simplified analytical model to evaluate the performance of the new architecture is described in Section 3. Section 4 details our Simulation environment, and the experimental basis for comparison of the new architecture with conventional load/store processors. The results obtained from our experiments are reproduced in this section. Section 5 describes the compiler optimizations that are essential to utilize the new architecture.

2. Our Architecture

The execution model of the proposed architecture relies on a non-blocking data-driven threads, akin to TAM [Culler 93] and Cilk [Blumofe 95] threads. At a programming level, a program is viewed as a set of activation frames and each frame consists of several non-blocking threads. A non-blocking thread typically corresponds to a basic block and an activation frame is typically a loop iteration or a function. The threads within an activation frame can share memory at cache level, have access to the registers of other threads, and in general can share the "state of the activation frame". The scheduling, data sharing and synchronization among the threads within an activation frame is treated differently from such activities among threads belonging to different activation frames. This leads to multi-level synchronization and scheduling of threads.

In our architecture, threads are non-blocking, and during execution they require *no memory accesses* -- all data

accesses are performed by "pre-loads" and "post-stores" (we will call our architecture as PL/PS, pronounced as PuLPS). Each thread is enabled when the required inputs are available (i.e., data driven model at a coarser grain). The number of inputs for a thread will be fixed -- typically to a subset of the number of registers available to a thread. Once enabled, a thread executes to completion where the instructions belonging to a thread will execute on a conventional pipelined CPU (with no memory access).

The results of a thread which are normally destined to other threads are not immediately supplied to waiting threads. The results of a thread are accumulated until the completion of the thread, to eliminate any delays and interruptions of the thread execution resulting from cache misses and other memory latencies during stores. Upon the completion of the thread execution, the results are handled by a separate hardware unit that executes "post-stores" to distribute the data to waiting threads. Whenever possible, the data can be delivered into the hardware registers of the destination threads directly. Compile time analysis can determine the target registers for the delivery of data. The decoupling of memory accesses from the thread execution eliminates thread stalls due to memory access (or thread-switches on cache misses [Agrawal 95]), and permits us to independently explore issues related to data distribution across memory hierarchies.

Simultaneous Multithreading [Tullsen 95, 96] of multiple active threads is inherent in our design. Such interleaving will permit us to maintain longer and sustained pipeline flows. However, the interleaving is limited to the threads of an activation frame, and compiler can generate the necessary code to control the interleaving.

Compiler support and program analysis at many levels in developing efficient non-blocking threads is crucial for the success of the new architecture. At a high level, the compiler must appropriately partition the source program into threads; at a low level, registers must be scheduled so as to avoid cache references or even costlier trips to primary storage. Through our use of SUIF on real programs, our future research will consist of applying user annotations, compile-time restructuring, and run-time specialization to obtain high performance on those portions of a program that are computationally intensive.

2.1. A block diagram of the architecture.

The overall block diagram of our architecture shown below (Fig. 1). The Scheduler performs all memory accesses (Pre Loads and Post Stores into available register contexts), and delivers the contexts to Activation Frame manager, which is responsible for selecting (a subset of) threads for execution on the pipeline. The Pipeline Control unit will interleave and execute instructions of the enabled threads on conventional pipelines (with no memory access). In this new architecture, both the memory pipeline and execution pipleline are simpler than conventional pipelines.

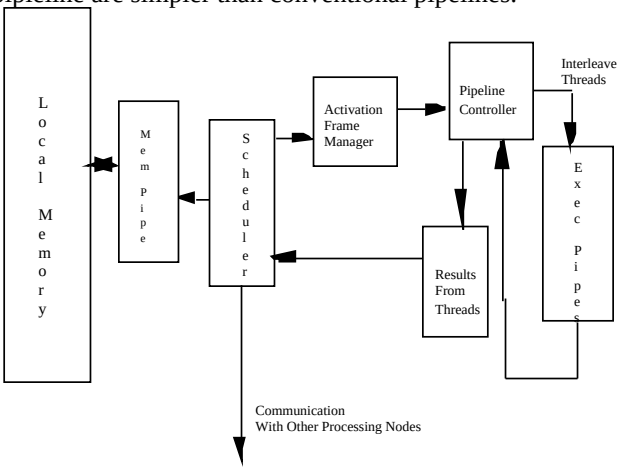

Fig. 1 PL/PS Block Diagram

3. Analytical Model evaluating the new architecture.

As a first step in investigating the potential improvements that can be gained from our proposed architecture, we have developed a simple analytical model. For the purpose of comparison, we will start with a traditional pipelined architecture, where load and store instructions are executed in the same pipeline. Consider for example a MIPS R4000 like pipeline.

The CPI can be expressed as;

= 1 + (Average number of stalls due to data dependencies) +(Average number of stalls due to memory accesses)

$= 1 + S_{dc} + S_{mc}$

where S_{dc} is the average number of stalls due to data dependencies and S_{mc} is the average number of stalls due to memory accesses. The stalls due to data dependencies include dependencies among arithmetic instructions as well as dependencies between load/store and arithmetic instructions. The stalls due to memory accesses depend on the average number of memory references, cache miss rate and cache miss penalty. Current architectures basically eliminate all stalls due to data dependencies among arithmetic instructions, and branch instructions using such techniques as result forwarding, delayed branches and branch prediction. However, dependencies that exist between memory access and arithmetic instructions do cause stalls.

The CPI for the new architecture is impacted by the following factors.

1. Stall in the pipeline. As with traditional pipelines, data dependencies among instructions could lead to stalls. However, the new architecture eliminates data dependencies among load/store instructions and arithmetic instructions, since all loads are completed before the arithmetic instructions are started (using the Preloads), and all stores become post-stores.

2. There could be delays in starting instructions in the pipeline when the preloads cannot be completely overlapped with the pipeline execution. This could happen when there are insufficient hardware contexts, insufficient thread level parallelism, memory bandwidth is insufficient to accommodate the preload and post-stores, or if the synchronization (or data dependencies) among the threads serializes the execution of the threads. In addition, cache misses could delay the preload and post-stores of thread contexts. However, it is our claim that the preload and post-stores provides for compile time optimizations of "blocking" thread contexts to permit the use of "Load Multiple" and "Store Multiple" instructions.

When there is a perfect overlap of thread execution in the pipeline with the preload and post-stores, the PL/PS architecture eliminates all stalls due to memory access encountered by instructions. CPI is given by $1 + S_{dn}$ where S_{dn} is the average number of stalls due to data dependencies in the pipeline of the new architecture. As mentioned above, this will be smaller than S_{dc}, the average number of stalls in traditional execution pipelines (since the stalls due to load/stores are eliminated). When there is less than perfect overlap of preloads and post-stores with the execution of threads in the pipeline, the CPI can be estimated as: $1 + S_{dn} + S_{mn}$ where S_{mn} is the average number of due to imperfect overlap. In order to obtain a perfect overlap, there should be sufficient work processed by the pipeline while the contexts are pre-loaded and post-stored.

It is also possible to model the situation as a queueing system. We have used a closed network as shown below (Fig. 2), with a fixed number of threads (n) in the system. As threads complete execution in the pipeline, they enable new threads (waiting for synchronization) with a probability of (1-p). It is straightforward to derive solutions for throughput, utilization and average queue lengths of each sub-queue shown above. We are more interested in obtaining values for the probability of overlap between the memory accesses and pipeline execution. This is obtained by finding the probability that both the pipeline and the memory access servers are simultaneously busy.

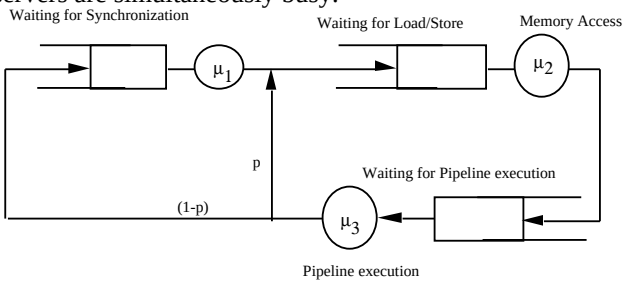

Fig. 1. A closed network of PL/PS architecture

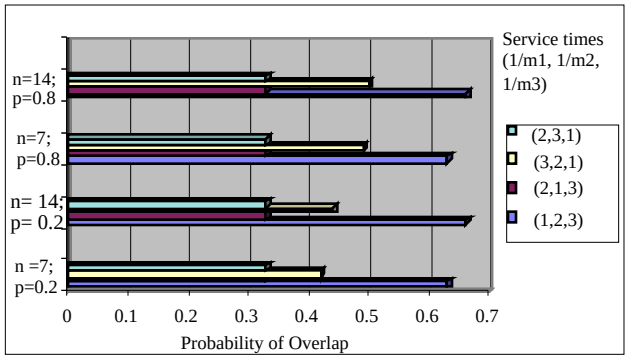

Figure 3. Probability of overlap analysis

The results are shown above (Fig. 3). The simple analysis indicates that when the delay due to synchronization is small (relative to the memory access delays and pipeline execution delays), we can achieve significant overlap between the memory and pipeline units, supporting our intuition. It should also be noted, that slower unit (either memory or pipeline) will throttle the flow through the system, by starving other units (increasing their idle times).

4. Simulation Environment and Results

To further analyze proposed architecture, we have developed a Monte Carlo simulation environment. In the simulation, threads with a random number of load, store and pipeline cycles are generated. A set of initial threads are processed by the Scheduler (Memory accesses) for Pre Loading register contexts of threads. These threads are then handed over to Pipeline control unit (with a preset context switching overhead). The pipeline unit will process threads (executes the pipeline cycles), overlapping with the Scheduler that may be preloading or poststoring data for other threads. When a thread completes execution in the pipeline, the thread context is placed in the Scheduler queue for poststoring its results (with a fixed context switch overhead). When the poststoring is completed by the Scheduler, additional threads are randomly enabled (created) and placed in the Scheduler queue, modeling the delivery of results to waiting threads (and enabling some of them). Fig. 4 represents the simulator diagramatically.

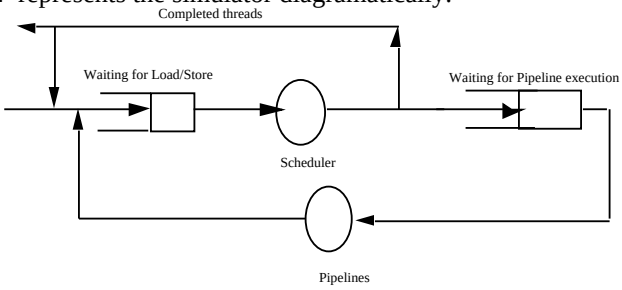

Figure 4. Simulation model of PL/PS architecture

In order to compare our architecture with conventional system, we have simulated a system with a single conventional processor that does both memory accesses (loads and stores) and executes non-memory pipeline

instructions. Completing threads enable (create) new threads as before. In order to make fair comparison, we have repeated our studies with multiple conventional processing units. However, when multiple units are used, we have included the potential for memory conflicts, which is set to be proportional to the number of memory accesses (loads and stores) needed by a thread.

For all systems, we have included the probability of a cache miss and cache penalty. Our intuition is that, when the memory latency increases (as a function of cache miss rate and miss penalty), the PL/PS architecture achieves better performance than conventional pipelines. In our experiments, we have varied the total number of threads processed, the average number of load, store, pipeline cycles per thread (i.e., thread granularity), the maximum number of register contexts with the pipeline and the memory unit (Scheduler), and the cache miss penalty. Finally, we varied the number of processing units in the conventional architecture, as well as the number of memory units (Scheduler) and pipelines in the PL/PS architecture. We describe our findings below. The results reported in this paper uses uniform distribution with a specified minimum and maximum value for all parameters (e.g., load, store, pipeline cycles, interarrival times).

4.1. Results of the Simulation Studies.

4.1.1. Total Execution Times. As a first step, we have randomly generated several threads to be processed by the new architecture, conventional architecture with a single processor and conventional architecture with 2 processors. Fig. 5 shows the results.

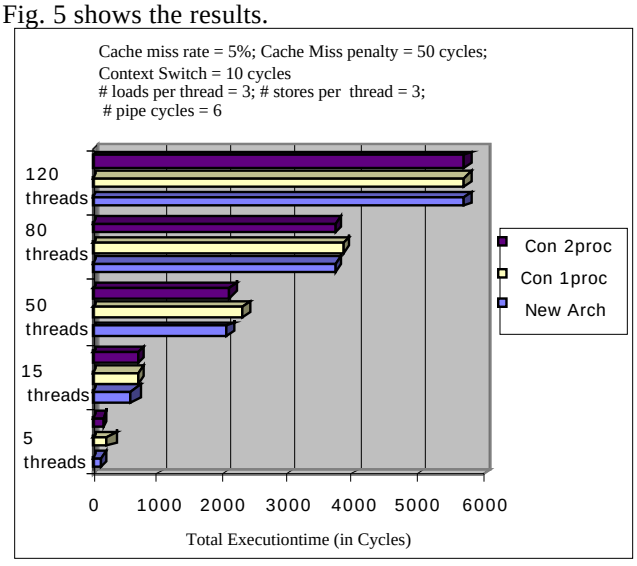

Figure 5. Total execution times under identical workloads

The total execution time taken to process the threads by PL/PS (new architecture) is consistently smaller than conventional architecture with a single processing unit. The performance conventional architecture with two processors matches that of PL/PS architecture. It should be noted, however, the two processors of conventional architectures are complete processing units (hence more hardware) while the two hardware units (viz., Scheduler and Activation Frame Manager) of the new architecture are not (hence less hardware). These results are obtained for a system with light load (the average queue lengths of waiting jobs is approximately 2, for all the systems examined).

We then explored the impact of heavier loads. Even when the average queue lengths exceeded 15, the trends are very similar (except for longer average waiting delays). Occasionally, the conventional architecture with 2 processors fared slightly better than the new architecture. The above results were generated using the same set of random threads (with random arrival times) for all 3 systems. However, since we propose to use a non-blocking thread model, where completing threads enable other waiting threads, we have repeated the experiment by randomly enabling new threads as threads exit the system, based on the individual systems' capabilities[1]. The results with a moderate loads (average number of jobs waiting for a processor is about 10) are shown Fig. 6.

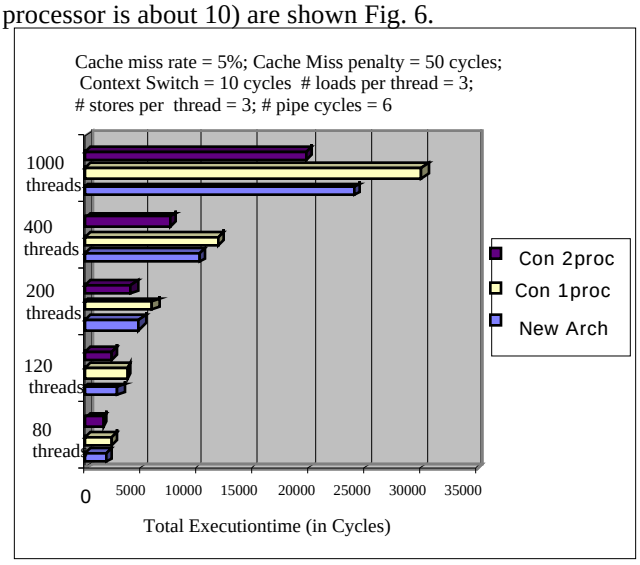

Figure 6. Total execution for non-blocking data driven threads

As can be seen from the diagram, PL/PS still outperforms conventional architecture with one processor; however, the conventional architecture with 2 processors performs better than PL/PS. This should be expected: with 2 processors, the delay due to synchronization (enabling of new threads) is smaller, which in turn reduces the overall waiting times and response times for jobs. Fig. 7 shows the average response times under moderate loads. Once again, PL/PS outperforms the conventional architecture with one

[1] In the previous workload, the times when a thread becomes enabled is the same for all 3 systems. In the new workload, the time when a new thread enabled depends on when a previous thread completes execution.

processor, but does not compare well with a 2 processor system.

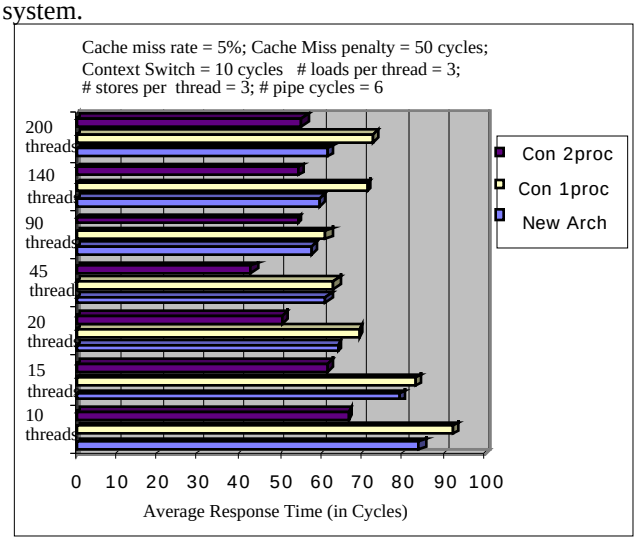

Figure 7. Average response times

4.1.2. Context-Switching Overhead. In the above experiments, we set the cost of passing thread contexts between the Scheduler and the Activation Frame Manager (in the PL/PS) to 10 cycles. Conventional architecture occurs no such context switching overhead. We feel that, with multiple hardware contexts for threads, the context switching overhead will be very small. The overall execution time for the PL/PS increases linearly with the context switching overhead.

4.1.3. Number of Hardware Contexts: We have also investigated the performance of PL/PS by restricting the number of available hardware contexts. The performance degrades significantly, only when the number of available hardware contexts drops below . Eight hardware contexts (8 for Scheduler and 8 for Activation Frame Manager) appeared to be sufficient for workloads used in our studies. Even under heavily loaded conditions, the average number hardware contexts reaches a balance, since new threads are enabled for execution only when other threads complete their execution (data driven scheduling). This finding is similar to our finding using the analytical model described in the previous section (Section 3).

4.1.4. Memory Latency. Our contention is that the PL/PS architecture out performs the conventional architecture, either with one or two processors, for longer memory latencies. In our simulations, the memory latencies are modeled using cache miss rates and cache miss penalties. We use the same rates and penalties for all 3 systems. Our intuition is supported by the data in Fig. 8; the new architecture performs better than the conventional architecture with 1 or 2 processors, when the miss penalty is moderate large (100). As the cache miss penalty becomes very large, all systems suffer performance degredation since the pipelines are idle (PL/PS will have imperfect overlap between memory and pipleine processors). In such cases, it may be worthwhile considering (context) switching to other ready threads (similar to the strategy used in Alewife [Agrawal 95]). However, since our model is based on non-blocking threads, we have not explored such strategies.

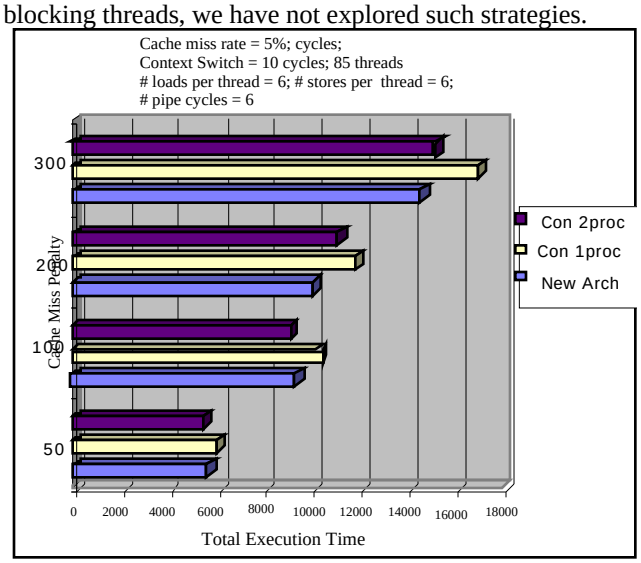

Figure 8. Significance of memory latency on overall execution time

4.1.5. Memory Conflicts. In the two processor (conventional system), we can vary the probability of a conflict for memory (since two autonomous processors are accessing memory to process the threads), and the delay due to the conflict. In all of the previous experiments, the conflict probability is set to 10% and the conflict penalty to 10 cycles. It should be clear that the two processor system will perform poorly with higher memory conflict rates and penalties (Fig. 9).

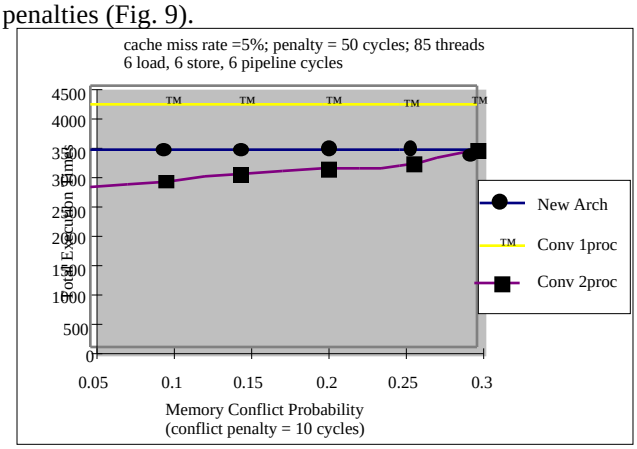

Figure 9. Impact of conflict for memory accesses on total execution time

4.1.6. Task Granularity. An additional parameter that can be varied is the granularity of a thread. In the above experiments, we have used threads with an average of 6 cycles of execution through the pipeline (6 load and store cycles per thread). These numbers are used to reflect the fine grained nature of our thread model. However, we feel that the

PL/PS architecture performs better than conventional architecture (with one processor) even for medium grained to large grained threads. This is partially supported by the data in Fig. 10.

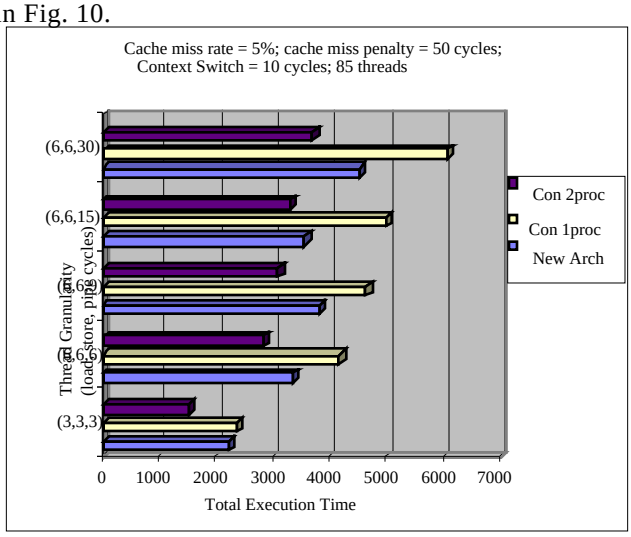

Figure 10. Impact of thread granularity on total execution time

4.1.7. Superscalars. Finally, we wanted to test if it is beneficial to add more memory processors (Scheduler) or more pipeline units (Activation Frame Managers). The data in Fig. 11 shows that it is better to add more memory processors. This is appealing given that we were attempting to tolerate memory latencies - more memory processors effectively reduce the memory latency. Fig. 11 also includes data for 1, 2 or 3 conventional processors processing the same workload. The work load consists of 85 threads; the test runs varied the granularity of threads from 9 cycles to 32 cycles per thread..

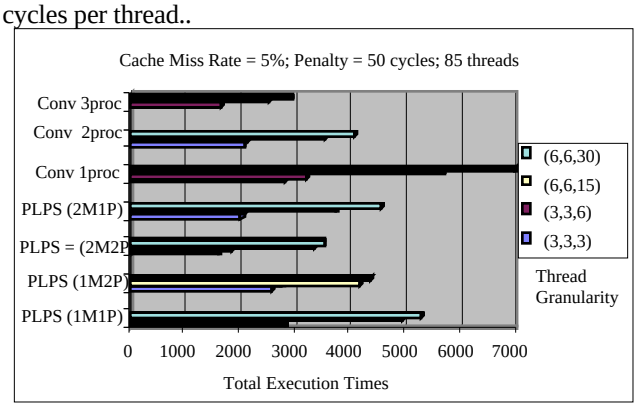

Figure 11. Trade-off analysis between multiple memory and pipeline units

4.1.7. Summarizing. Our experiments support our intuition that the overlapped access to memory with the execution of non-memory instructions will improve the performance of multithreaded architectures. While conventional architectures with 2 processors performs as well as (and sometime better than) the PL/PS architecture, one needs to remember that the 2 processors implies more complex hardware than PL/PS and its performance is sensitive to the memory conflict probability and conflict penalties. While the non-blocking, data driven model advocated implies fine grained threads, the PL/PS architecture also does well for medium grained threads. The experiments are based on randomly generated threads. Some of the workload parameters (e.g., thread granularities) are based on a cursory examination of code generated from simple Cilk programs.

5. Conclusions and Future Research

In this paper we presented a new architecture that decouples memory accesses from pipeline execution. The decoupling of memory accesses from thread execution requires a separate unit to perform the necessary pre-loads and post-stores, and controlling the allocation of hardware contexts to enabled threads. This separation facilitates for achieving high locality and minimizing the impact of distribution and hierarchy in large memory systems. The non-blocking nature of threads eliminates the need for thread switching (resulting from synchronization requirements), thus improving the overhead in scheduling of threads.

Our preliminary results are very encouraging. The PL/PS architecture always outperforms conventional architecture with one processor. It also performs better than a 2 processor (conventional architecture), when the probability of memory conflicts and memory latency become significant. Our studies also indicate that multiple pipelines (i.e., superscalars) are less beneficial without multiple hardware units accessing memory in a multithreaded architecture.

Our studies have assumed a non-blocking thread model. While there are programming languages that support such a model (e.g., Cilk), it may be worthwhile considering compiler technology to automatically convert traditional blocking threads into non-blocking threads. In our studies we have assumed that it is possible to generate sufficient number threads to achieve an almost perfect overlapped execution between the memory access and pipeline execution. Instructions of a thread must be reordered with all loads at the beginning and all stores at the end of thread execution. Such reordering for conventional programming models will produce very fine grained threads that may cause excessive context switching overheads. Aggressive compiler analyses are needed to generate optimal threads with medium grained threads (with longer run-lengths). This may necessitate new approaches to register allocation (e.g., it may be beneficial to load a value into multiple registers, in different contexts; overlapping the input registers of a thread with the result registers of another), predictive preloading across branches based on branch prediction techniques, and data placement (e.g., blocking the variables of a thread would permit “load multiple” type instructions to efficiently preload a thread context).

While we advocated multithreading as the solution to memory latency, other researchers have been exploring different solutions, including Data scalar [Berger 97], Multiscalar [Sohi 95], processor-memory integration [Saulsbury 96], and aggressive prefetching techniques [Baer 91].

Acknowledgement. This research is supported in part by the following grants from NSF, MIPS-9622593, MIP-9622836, CDA-9529561.

6. References

[Agrawal 95] Agrawal, A., et. al. "The MIT Alewife machine: Architecture and performance", Proc. of 22nd Intl. Symp. on Computer Architecture (ISCA-22), 1995, pp 2-13.

[Alverson 90] Alverson, R. *et al.*, "The Tera Computer System," *International Conference on Supercomputing*, 1990, pp 1-6.

[Baer 91] Baer, J-L. and Chen, T. "An effective on-chip preloading scheme to reduce data access penalty," *Proceedings of Supercomputing'91*, Nov. 1991, pp 178-186

[Berger 97] Berger, D., Kaxiras, S., and Goodman, J., "Datascalar architecture," *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

[Blumofe 95] Blumofe, R. D. *et. al.*, "Cilk: An efficient multithreaded runtime system", Proc of the *5th ACM Symposium on Principles and Practice of Parallel Programming (PPoP)*, July 1995.

[Culler 93] Culler, D. E, Goldstein, S. C., Schauser K. E. and von Eicken, T., "TAM- A compiler controlled threaded abstract machine," *Journal of Parallel and Distributed Computing*, 18, pp 347-370.

[Fang 97] Fang, K. "In-place updates and other I-structure cache optimizations", Ph.D. dissertation (in preparation) Dept. CSE, UTA, Arlington, TX 76019-0015, Expected completion date Dec. 1997.

[Kavi 95] Kavi, K.M. et. al. "Design of cache memories for multi-threaded dataflow architecture", *Proceedings of the 22nd International Conference on Computer Architecture*, pp 253-264, May 1995.

[Kavi 96] Kavi, K.M. and Hurson, A.R. "Investigation of operand memory reuse in a dynamic dataflow architecture", *Proceedings of the High Performance Computing Symposium 96,* (The society of computer simulation), pp 288-295, April 8-11, 1996, New Orleans, Louisiana.

[Kavi 97] Kavi, K.M. and Hurson, A.R. "Performance of cache memories in dataflow architectures", To appear in the *Euromicoro Journal on Systems Architecture.*

[Pier 83] Pier, K. A., "A retrospective on the Dorado. A high performance personal computer," *Proceedings of the 10th International Symposium on Computer Architecture*, pp 252-269, 1983.

[Saulsbury 96] Saulsbury, A., Pong, F., and Nowatzyk, A., "Missing the memory wall: A case for processor/memory integration," *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996, pp 90-101.

[Smith 85] Smith, B., "The architecture of HEP" in *Parallel MIMD Computation: HEP Supercomputer and applications*, edited by J. S. Kowalik, MIT Press 1985.

[Sohi 95] Sohi, G, Breach, S, and Vijaykumar, T., "Multiscalars processors", *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995, pp 414-424.

[Tullsen 95] Tullsen, D. M., *et al.*, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995, pp 392-403.

[Tullsen 96] Tullsen, D. M., *et al.*, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," *Proceedings of the 23rd International Symposium on Computer Architecture*, 1996.