A Realistic Study on Multithreaded Superscalar Processor Design

Yuan C. Chou, Daniel P. Siewiorek, and John Paul Shen

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.
{yuanchou,siewiorek,shen}@ece.cmu.edu

Abstract. Simultaneous multithreading is a recently proposed technique in which instructions from multiple threads are dispatched and/or issued concurrently in every clock cycle. This technique has been claimed to improve the latency of multithreaded programs and the throughput of multiprogrammed workloads with a minimal increase in hardware complexity. This paper presents a realistic study on the case for simultaneous multithreading by using extensive simulations to determine balanced configurations of a multithreaded version of the PowerPC 620, measuring their performance on multithreaded benchmarks written using the commercial P Threads API, and estimating their hardware complexity in terms of increases in die area. Our results show that a balanced 2-threaded 620 achieves a 41.6% to 71.3% speedup over the original 620 on five multithreaded benchmarks with an estimated 36.4% increase in die area and no impact on single thread performance. The balanced 4-threaded 620 achieves a 46.9% to 111.6% speedup over the original 620 with an estimated 70.4% increase in die area and a detrimental impact on single thread performance.

1 Introduction

Simultaneous multithreading is a recently proposed technique in which instructions from multiple threads are dispatched and/or issued concurrently in every clock cycle. To aggressively exploit instruction-level parallelism on a wide-issue superscalar, a large number of functional units are necessary to achieve good performance even though the average utilization of most of the functional units is low. This is due to the unsmooth parallelism profile of most programs. By executing instructions from multiple threads concurrently, simultaneous multithreading has the potential effect of smoothing the aggregate parallelism profile and thereby making better use of the functional units to improve the latency of a multithreaded program or the throughput of a multiprogrammed workload, making it an attractive microarchitectural technique.

Recent studies in simultaneous multithreading [9, 10] have demonstrated an average throughput of 5.4 instructions per cycle (IPC) on a suite of multiprogrammed workloads each comprising of 8 SPEC benchmarks. While these results are very promising, the chosen workloads may not be very representative of realistic environments. Other studies [6, 7, 8, 14] have demonstrated the potential of simultaneous multithreading on multithreaded applications. However, these research have assumed the use of a custom threading paradigm. We believe that for simultaneous multithreading to be widely adopted in industry, it must be able to take advantage of the multithreaded programs that have been written for symmetric multiprocessors (SMPs). These programs are usually written in a threads package supported by the operating system. Current operating systems such as OSF/1, Mach, Solaris, AIX, Windows NT and

Windows 95 all support multithreading and provide programmers with threads application programming interfaces (APIs).

This paper attempts to present a more realistic examination of the case for simultaneous multithreading by evaluating the performance of realistic simultaneous multithreaded superscalar processor models on multithreaded applications written using a commercial threads API and estimating the complexity required to implement these processors in terms of die area. Towards this end, we choose a real superscalar processor, the PowerPC 620 [1], as the baseline microarchitecture, and the IBM AIX 4.1 P Threads API, which is based on the emerging POSIX 1003.1c industry standard for a portable user threads API, as the threads API for creating our multithreaded applications. In the course of our study, we also attempt to understand the bottlenecks of a realistic simultaneous multithreaded processor and answer questions such as the number of threads the processor should support, the fetch and dispatch strategy the processor should employ and the functional unit mix the processor should adopt in order to achieve an efficient and balanced design. This is in contrast to previous studies in which execution resources are predetermined (usually arbitrarily expanded from a non-multithreaded design) before support for simultaneous multithreading is added to this baseline processor.

The rest of this paper is organized as follows. Section 2 gives an overview of the methodology employed in this research, as well as a description of the simulation environment and the benchmarks used. Section 3 describes the PowerPC 620 and the parameters studied in designing a balanced multithreaded 620. Section 4 presents the effects of varying these parameters and the resulting configurations of balanced multithreaded 620s while Section 5 presents an estimation of the extra complexity required to implement these processors. Section 6 briefly mentions related work. Finally, Section 7 concludes this paper.

2 Methodology

To evaluate the performance of simultaneous multithreaded superscalar processor models, we first identify the resources of the baseline PowerPC 620 that have to be replicated or modified in order to support simultaneous multithreading, and we also identify the major parameters that affect the performance of the multithreaded processor. The PowerPC 620 is chosen as the baseline microarchitecture because it represents the current state-of-the-art in superscalar processor design and because we already have a cycle-accurate trace-driven simulator for this microarchitecture [2]. Simulations are then performed using five multithreaded benchmarks written using the IBM AIX 4.1 P Threads API and based on the simulation results, we identify balanced configurations of the multithreaded processors.

2.1 Simulation Environment

We modified the original PowerPC 620 simulator to incorporate the additional features required to support simultaneous multithreading. In addition, we also developed a tool that automatically generates PowerPC traces from multithreaded C/C++ programs written using the IBM AIX 4.1 P Threads API. This trace generation tool works by instrumenting the assembly files of the original C/C++ multithreaded program. In addition to generating the instruction and data traces, it also generates additional thread synchronization information that is used by the simulator to ensure that the synchronization structure of the original multithreaded program is preserved during simulation.

2.2 Benchmarks

Five multithreaded benchmarks representing CAD, multimedia and scientific applications are used in the simulations. We believe that these applications are realistic workloads for current and future high performance computers and therefore are good target applications for simultaneous multithreaded processors. The first three are integer benchmarks while the last two are floating-point benchmarks. All benchmarks are simulated till completion. These benchmarks are briefly described in Table 1.

TABLE 1. Description of benchmarks.

Benchmark	Description	Dynamic Inst Count
Router	Multithreaded version of ANAGRAM II [3].	14.1M
MPEG	Parallel decoding of four MPEG-1 video streams.	26.4 M
Matrix Multiply	Parallel integer matrix multiply [4].	5.1 M
FFT	P Threads implementation of SPLASH-2 [5] FFT.	22.9 M
LU	P Threads implementation of SPLASH-2 [5] LU.	16.6 M

3 The Multithreaded PowerPC 620

The PowerPC 620 [1,2] is used as the baseline processor in developing our simultaneously multithreaded processor models.

3.1 Replicated Resources

In the P Threads API, a thread is defined as a basic unit of CPU utilization that consists of a program counter, a register set, and a stack space, that shares its address space with other peer threads. To allow instructions from multiple threads to exist in the processor at the same time, the program counter, the GPR and FPR files, the Condition Register (CR), the Link Register as well as its Link Register Stack and Shadow Link Registers, the Count Register as well as its Shadow Count Register are replicated. To simplify branch misprediction and exception recovery, the reorder buffer is also replicated.

Although the GPR, FPR and CR rename buffers can be shared among threads, this will result in a multiplicative increase in the number of read and write ports to these buffers which can seriously impact machine cycle time. Therefore, we choose to replicate these rename buffers rather than to share them among threads.

Depending on the instruction fetch strategy employed, the number of instruction cache ports may or may not need to be increased. The instruction queue can be replicated or shared among threads. We choose to replicate it because it reduces the complexity of the dispatch mechanism.

3.2 Shared Resources

The functional units and their reservation stations are shared by all threads, as are the instruction and data caches and their memory management units. The bus interface unit and the L2 cache interface are also shared by all threads. Since all threads share the same code section, the branch target address cache (BTAC) is shared. We also choose to share the branch history table (BHT). In Section 4.5, we show that simultaneous multithreading has little impact on the hit rates of the shared caches and the prediction accuracy of the BHT.

3.3 Variable Parameters

Having identified the replicated and shared resources, we identify four major parameters that can be varied to achieve a balanced multithreaded 620. They are: 1) the number of threads supported, 2) the instruction fetch strategy and bandwidth, 3) the instruction dispatch strategy and bandwidth, and 4) the functional unit mix. The effects of varying these parameters are presented in the next section.

4 Experimental Results

In order to prune the design space to a manageable size (which is necessary since each simulation run takes about 2 days to complete on our fastest CPUs), we adopted the following design space exploration methodology. We first examine the effects of different fetch and dispatch strategies while assuming an expanded functional unit mix, and we select combinations that provide good cost-performance tradeoffs. Next, assuming these selected instruction fetch and dispatch strategies, we gradually scale back the functional unit mix, eliminating those units that do not contribute to better performance. As the number of instructions executed by each benchmark remains constant for all processor configurations, the results are presented in terms of the average number of instructions completed per cycle (IPC). We do not attempt to quantify the impact of our microarchitectural changes and additions on machine cycle time. In all the tables presented in this section, the numbers shown are IPC numbers. "1T" means that configuration supports one thread (i.e. it is non-multithreaded) while "2T" and "4T" represent configurations that support two and four threads respectively.

4.1 Number of Threads

In Section 3.1, the resources that have to be replicated for each thread supported by the processor are described. In addition, the number of threads supported also has a direct impact on the number of buses connecting the functional units to the rename buffers and reorder buffers. Moreover, since the main advantage of a multithreaded processor over a multiprocessor is its ability to share functional units to reduce the amount of resources required to achieve the same performance level, increasing the number of threads supported increases the number of replicated resources relative to the number of shared resources, thus diminishing this advantage. For this reason, in our study, we only consider supporting two threads and four threads. The one threaded case is included for comparison purposes.

4.2 Fetch and Dispatch

The number of threads from which instructions are fetched in each cycle (F) can be varied, from fetching from one thread to fetching from all threads. Although the number of instructions fetched from a thread can also be varied, we choose to retain the 620's design of fetching at most four instructions per thread. To fetch from multiple threads in each cycle, we model the instruction cache as being multi-ported. In terms of dispatch strategy, we assume that instructions are dispatched from all threads in every cycle and we vary the number of instructions dispatched from each thread per cycle (**D**).

TABLE 2. Functional unit configuration of the baseline 620.

Functional Unit Type	#Units	#Reservation Station Entries	Issue Latency
Single-Cycle Integer Unit (SCIU)	2	2	1
Multi-Cycle Integer Unit (MCIU)	1	2	3-8 (multiply) 37 (divide)
Floating Point Unit (FPU)	1	2	1 (multiply-add) 18 (divide) 22 (square root)
Load/Store Unit (LSU)	1	3	1
Branch Unit (BRU)	1	4	1

The functional unit mix of the original 620 is shown in Table 2. In studying the effects of our fetch and dispatch combinations, we double the number of SCIUs, MCIUs and FPUs, and we add a second execution pipeline to both the LSU and the BRU. Since simultaneous multithreading increases the dispatch rate, we double the number of reservation station entries of each functional unit. The number of SCIUs, MCIUs, and FPUs can easily be increased but not the LSU. This is because of the cost and complexity of having multiple data cache ports and because of the complexity of the 620's alias detection mechanism (the 620 allows loads and stores to execute out-of-order). The second execution pipeline allows up to two loads or two stores to be issued and executed in every cycle. In the enhanced BRU, the two execution pipelines share a common reservation station. Within a thread, branch instructions are issued in-order.

TABLE 3. Effects of varying instruction fetch and dispatch strategy (IPC).

	Router			MPEG			Matrix Multiply			FFT			LU		
<F, D>	1T	2T	4T	1T	2T	4T	1T	2T	4T	1T	2T	4T	1T	2T	4T
<1, 2>		1.80	1.95		2.54	2.99		1.63	1.81		1.73	2.04		1.75	1.80
<1, 4>	1.29	1.92	1.98	1.89	2.67	3.00	1.19	1.61	1.74	1.18	1.75	2.08	1.11	1.75	1.79
<1, 8>	1.30	1.93		1.89	2.66		1.20	1.61		1.18	1.76		1.11	1.76	
<2, 2>		1.83	**2.19**		2.63	**3.60**		1.62	**1.80**		1.76	**2.25**		1.76	**1.96**
<2, 4>		**1.97**	2.26		**2.91**	3.69		**1.61**	1.77		**1.78**	2.27		**1.77**	1.97
<2, 8>		1.97			2.93			1.61			1.80			1.78	
<4, 2>			2.24			3.62			1.82			2.26			1.97
<4, 4>			2.32			3.73			1.75			2.28			1.97

F = number of threads instructions are fetched from in every cycle
D = number of instructions dispatched per thread in every cycle

Table 3 shows the effects of the various fetch and dispatch combinations on the performance of the multithreaded 620. For the 2T processor, fetching from two threads in every cycle and dispatching 4 instructions/thread (<F, D> = <2, 4>) is a good trade-off. For the 4T processor, if the total dispatch bandwidth is limited to 8 instructions/cycle (i.e. dispatch 2 instructions/thread/cycle), fetching from 2 threads/cycle almost matches the performance of fetching from 4 threads/cycle except on the *Router* benchmark. Increasing the total dispatch bandwidth to 16 instructions/cycle results in a moderate performance gain on the *Router* and *MPEG* benchmarks. However, dispatching 16 instructions/cycle may be unrealistic given current technology. Therefore, we conclude that for the 4T processor, fetching from two threads every cycle and dispatching 2 instructions/thread (<F, D> = <2, 2>) is a good trade-off. Finally, we note that dispatching 8 instructions/thread/cycle for both the 1T and 2T processors results in little performance gain because we limit the fetch bandwidth to 4 instructions/thread. Fetching more than 4 instructions/thread may require fetching from multiple cache lines (since the typical basic block length is 4-5 instructions for integer programs), thus demanding more sophisticated fetching mechanisms [15].

4.3 Functional unit Mix

In our study of instruction fetch and dispatch strategies, we have deliberately assumed a rich functional unit mix. However, not every one of the functional units may be well utilized and the number of functional units may possibly be scaled back with little or no sacrifice in performance.

Table 4 shows the effects of scaling back the number of functional units. Case A is the expanded functional unit mix assumed earlier. In Cases B and C, we scale back the number of SCIUs by one and two units respectively. In Cases D and E, we remove the second MCIU and the second FPU respectively. In Case F, we remove the second LSU execution pipeline, while in Case G, we remove the second BRU execution pipeline. Examining this table, we observe that for both the 2T and the 4T processors, removing the second LSU execution pipeline and removing two SCIUs each results in significant performance degradation, while eliminating the second BRU execution pipeline has little impact on performance. Eliminating the second MCIU only degrades the performance of the 4T processor on the *Matrix Multiply* benchmark. Removing the second FPU degrades the performance of both the 2T and the 4T processors on the *FFT* benchmark by less than 4%. Since a FPU is expensive to implement (see Table 6), the second FPU is difficult to justify.

Based on these observations, in Case H (<S, M, F, L, B> = <3, 1, 1, 2, 1>), we eliminated the second MCIU, the second BRU execution pipeline, the second FPU as well as the fourth SCIU, resulting in a functional unit mix that is a good trade-off for both the 2T and the 4T processors. For comparison purposes, in Case I, we set the functional unit mix and number of reservation entries to that of the original 620. The performance of the 2T and 4T processors are much lower than in Case H, indicating that the functional unit mix and reservation station entries of the original 620 must be suitably enriched for simultaneous multithreading to be effective. At this point, we make two observations. First, among functional units, the LSU is a bottleneck in the multithreaded processors. In fact, when the rich functional unit mix of Case A is assumed, our simulation statistics show that the dual execution pipeline LSU is the most saturated functional unit of both the 2T and 4T processors. Second, in addition to the second LSU execution pipeline, only an additional SCIU need to be added to the functional unit mix of the 620 for effective multithreading. This indicates that simultaneous multithreading is making more efficient use of the execution resources due to the smoothing of the aggregate parallelism profile.

TABLE 4. Effects of varying functional unit mix (IPC).

Case	Router			MPEG			Matrix Multiply			FFT			LU		
<S, M, F, L, B>	**1T**	**2T**	**4T**	**1T**	**2T**	**4T**	**1T**	**2T**	**4T**	**1T**	**2T**	**4T**	**1T**	**2T**	**4T**
A <4, 2, 2, 2, 2>	1.29	1.97	2.19	1.89	2.91	3.60	1.19	1.60	1.80	1.18	1.78	2.25	1.11	1.77	1.96
B <3, 2, 2, 2, 2>	1.29	1.96	2.19	1.88	2.86	3.58	1.19	1.61	1.82	1.18	1.78	2.24	1.11	1.78	1.96
C <2, 2, 2, 2, 2>	1.29	1.92	2.16	1.84	2.63	3.09	1.19	1.61	1.80	1.17	1.75	2.21	1.11	1.78	1.94
D <4, 1, 2, 2, 2>	1.29	1.96	2.19	1.87	2.89	3.58	1.19	1.60	1.65	1.18	1.78	2.25	1.10	1.76	1.95
E <4, 2, 1, 2, 2>	1.29	1.96	2.19	1.89	2.91	3.60	1.19	1.61	1.80	1.14	1.72	2.17	1.11	1.78	1.92
F <4, 2, 2, 1, 2>	1.24	1.86	2.04	1.77	2.69	3.24	1.13	1.59	1.66	1.15	1.72	2.12	1.07	1.57	1.62
G <4, 2, 2, 2, 1>	1.28	1.96	2.19	1.87	2.86	3.56	1.19	1.61	1.82	1.18	1.78	2.25	1.11	1.76	1.96
H <3, 1, 1, 2, 1>	1.28	**1.95**	**2.19**	1.85	**2.81**	**3.47**	1.19	**1.60**	**1.66**	1.14	**1.72**	**2.18**	1.10	**1.75**	**1.92**
I *Original 620*	1.20	1.77	1.82	1.64	2.24	2.67	1.16	1.57	1.64	1.08	1.51	2.00	1.06	1.55	1.52
<S, M, F, L, B> = <#SCIUs, #MCIUs, #FPUs, #LSU pipelines, #BRU pipelines>															

4.4 Comparison of Multithreaded 620 vs. Original 620

To evaluate the effectiveness of simultaneous multithreading, the performance of the balanced multithreaded 620s is compared to that of the original 620. The balanced 2-threaded (2T) 620 fetches instructions from both threads in every cycle, dispatches and completes up to four instructions per thread per cycle (<F, D> = <2, 4>). In addition to the baseline 620's functional unit mix, it has a dual-execution pipeline LSU and an additional SCIU (<S, M, F, L, B> = <3, 1, 1, 2, 1>). It also has twice as many reservation station entries in each functional unit. The balanced 4-threaded (4T) 620 fetches instructions from two threads in every cycle, dispatches and completes up to two instructions per thread per cycle (<F, D> = <2, 2>), and has the same functional unit mix and number of reservation station entries as the balanced 2T 620. Also included in the comparisons is an expanded but still single-threaded 620. The expanded 620 dispatches and completes up to four instructions per cycle (<F, D> = <1, 4>), and has the same functional unit mix and number of reservation station entries as the balanced 2T and 4T 620s. Table 5 presents the results of the comparison. The expanded 620 improves upon the performance of the original 620 by 3.8% to 12.8%. The balanced 2T 620 shows a 41.6% to 71.3% improvement, while the balanced 4T 620 shows a 46.9% to 111.6% improvement.

Although simultaneous multithreading improves the performance of multithreaded programs, it can also degrade the performance of single-threaded programs since the dispatch bandwidth is now partitioned among multiple threads. To highlight the impact of simultaneous multithreading on single-thread performance, the performance of the balanced 2T and 4T processors are measured on two SPEC92 single-threaded benchmarks, *eqntott* and *tomcatv*. As shown on the right half of Table 5, the 2T processor has the same performance as the expanded 620 since it dispatches 4 instructions/thread every cycle (like the original and expanded 620s) and has an identical functional unit mix. In contrast, because the 4T processor dispatches only 2 instructions/thread in every cycle, its performance on these single-threaded programs suffers. On the *eqntott* benchmark, its performance is 21.2% worse than the original 620. This leads to the observation that if separate instruction queues are assumed for each thread and the total dispatch bandwidth is limited to 8 instructions/cycle, the 2T processor is a better trade-off than the 4T processor when the target applications of the processor comprise both multithreaded and single-threaded applications.

TABLE 5. Balanced multithreaded 620s vs. original 620 on multithreaded and non-multithreaded benchmarks (IPC).

	Multithreaded Benchmarks					Single-threaded	
	Router	MPEG	Matrix Multiply	FFT	LU	eqntott	tomcatv
Original 620	1.20	1.64	1.13	1.08	1.06	1.32	1.00
Expanded 620	1.28 +6.7	1.85 +12.8	1.19 +5.0	1.14 +5.6	1.10 +3.8	1.35 +2.3	1.01 +1.0
Balanced 2-threaded 620	1.95 +62.5	2.81 +71.3	1.60 +41.6	1.72 +59.3	1.75 +65.1	1.35 +2.3	1.01 +1.0
Balanced 4-threaded 620	2.19 +82.5	3.47 +111.6	1.66 +46.9	2.18 +101.9	1.92 +81.1	1.04 - 21.2	0.99 - 1.0

4.5 Effects on Caches and Branch Prediction

We also studied the impact of simultaneous multithreading on the hit rates of the shared instruction and data caches and the prediction accuracy of the shared branch history table (BHT) when the multithreaded benchmarks are run. Due to limited space, we briefly state that simultaneous multithreading has negligible impact on the hit rate of the shared instruction cache. It degrades the hit rate of the data cache by less than 2% on all benchmarks for both the 2T as well as the 4T processor. It also has little impact on the shared BHT. The prediction accuracy for both the 2T and 4T processors are within 3% of that of the original 620.

5 Complexity Estimation

The commonly cited advantage of a multithreaded processor over a multiprocessor is that the former requires less resources to achieve a given level of performance because of the sharing of execution units. In this section, a first-order estimation of the extra complexity required to implement the balanced multithreaded 620s is made.

The estimation is made by determining from the die photo (not shown due to space limitations) of the original 620 the die areas of the various shared and replicated resources. We make the simplistic assumption that the area of the replicated resources scale linearly with the number of replications unless otherwise stated. Since the number of reservation station entries of the functional units are doubled in the 2T, 4T and Expanded 620s, we estimate the die areas of these functional units to be 25% larger. Although the instruction cache of the 2T and 4T processors are assumed to be dual-ported in our simulations, here we assume that they occupy the same die area as the data cache which is two-way interleaved. We assume that the die area of the enhanced LSU is twice that of the original LSU. We also assume that the die area occupied by the dispatch and completion unit of the 2T 620 is twice that of the original 620 while that of the 4T 620 is four times larger.

The results of the estimation are shown in Table 6. The units of area are relative square units. The balanced 2-threaded 620's die area is estimated to be 36.4% larger than that of the original 620, while that of the balanced 4-threaded 620 is estimated to be 70.4% larger. For comparison purposes, the expanded 620 is estimated to be 12.6% larger than the original 620. Overall, the die area increases appear to be more significant than implied in previous studies but is still reasonable.

TABLE 6. Comparison of die areas (in relative square units).

	Original 620	**Expanded 620**	**Balanced 2T 620**	**Balanced 4T 620**
Instruction Cache + MMU	26	26	31	31
Data Cache + MMU	31	31	31	31
Bus + L2 Interface + Perf. Monitor + PLL + COP JTAG	19	19	19	19
SCIU	8	10	15	15
MCIU	6	7.5	7.5	7.5
FPU	18	22.5	22.5	22.5
LSU	9	18	18	18
BRU	5	6.5	6.5	6.5
GPR and Rename Buffers	3	3	6	12
FPR and Rename Buffers	6	6	12	24
Dispatch + Completion Unit	16	16	32	64
Total	147	165.5 +12.6	200.5 +36.4	250.5 +70.4

6 Related Work

Other than the studies mentioned in Section 1, [11] studied the effects of limited fetch bandwidth and instruction queue size on a 2-threaded superscalar processor with unlimited issue width and unlimited functional units while [12, 13] studied simultaneous multithreaded processors that dynamically interleave VLIW instructions.

7 Conclusion

The cost/performance trade-offs of both the 2-threaded and 4-threaded processors when executing multithreaded benchmarks are reasonable although much less impressive than reported in previous studies using multiprogrammed workloads. The dispatch bandwidth and load/store unit limitations result in the diminishing returns of the 4T processor. Because of the partitioning of the dispatch bandwidth among multiple threads, the 4-threaded processor actually degrades the performance of non-multithreaded programs.

In conclusion, we believe that the future of simultaneous multithreading depends on the widespread adoption of the multithreaded programming paradigm. If this programming paradigm is widely adopted, resulting in a proliferation of multithreaded applications, simultaneous multithreaded processors are an efficient means of exploiting the thread and instruction level parallelism of these applications. On the other hand, if single-threaded applications continue to dominate or if future single-threaded processors are able to effectively utilize the available dispatch bandwidth, the future of simultaneous multithreading may be less promising.

References

[1] D. Levitan. T. Thomas, and P. Tu, "The PowerPC 620 Microprocessor: A High Performance Superscalar RISC Microprocessor", in *Spring CompCon 95 Proceedings*, pages 285-291, 1995.

[2] T. A. Diep, C. Nelson and J. P. Shen, "Performance Evaluation of the PowerPC 620 Microarchitecture", in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 163-175, 1995.

[3] J. M. Cohn, D. J. Garrod, R. A. Rutenbar, and L. R. Carley, "KOAN/ANAGRAM II: New Tools for Device-Level Analog Placement and Routing", in *IEEE Journal of Solid-State Circuits*, Vol. 26, No. 3, March 1991.

[4] J. Boykin, D. Kirschen, A. Langerman, and S. LoVerso, "Programming Under Mach", Addison- Wesley, 1993.

[5] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations", in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24-36, 1995.

[6] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa, "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads", in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 136-145, 1992.

[7] M. Gulati and N. Bagherzadeh, "Performance Study of a Multithreaded Superscalar Microprocessor", in *Second International Symposium on High-Performance Computer Architecture*, pages 291-301, 1996.

[8] M. Loikkanen and N. Bagherzadeh, "A Fine-Grain Multithreading Superscalar Architecture", in *Proceedings of PACT '96*, pages 163-168, 1996.

[9] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392-403, 1995

[10] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor", in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191-202, 1996.

[11] G. E. Daddis and H. C. Torng, "The Concurrent Execution of Multiple Instruction Streams on Superscalar Processors", in *International Conference on Parallel Processing*, pages I 76-83, 1991.

[12] R. G. Prasadh and C. Wu, "A Benchmark Evaluation of a Multi-Threaded RISC Processor Architecture", in *International Conference on Parallel Processing*, pages I 84-91, 1991.

[13] S. W. Keckler and W. J. Dally, "Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism", in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 202-213, 1992

[14] M. Bekerman, A. Mendelson, and G. Sheaffer, "Performance and Hardware Complexity Trade-offs in Designing Multithreaded Architectures", in *Proceedings of PACT '96*, pages 24-34, 1996.

[15] T. M. Conte, K. N. Menezes, P. M. Mills, and B. Patel, "Optimization of Instruction Fetch Mechanisms for High Issue Rates", in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 333-344, 1995.

Acknowledgments

This research is supported by the NSF under grant MIP-9403473 and by the ONR under Contract N00014-96-1-0347.