Explorations in Symbiosis on two Multithreaded Architectures

Allan Snavely*[†], Nick Mitchell*, Larry Carter*[†], Jeanne Ferrante*[†], and Dean Tullsen*

Abstract

Symbiosis is defined as the mutually beneficial living together of two dissimilar organisms in close proximity. We adapt that term to refer to the increase in throughput that can occur when two or more applications are executed concurrently on a multithreaded computer. In this paper, we give a formal definition of symbiosis, make observations about its nature, and present experimental results on two multithreaded architectures, the Tera MTA and (via simulation) a Simultaneous Multithreading machine.

1 Introduction

Multithreaded architectures such as SMT (Simultaneous Multithreading) [8] and the Tera MTA (Multi-Threaded Architecture) make use of multiple threads of execution to keep the processor highly utilized in the face of latencies. When one thread of execution has to wait for the result of an operation such as a memory reference, another thread with a ready-to-execute instruction can use the processor. There are, in general, two sources of threads for these machines; 1) threads resulting from the parallel decomposition of a single program and 2) threads from different programs. On the SMT and on the MTA, programs from a multiprogrammed environment become a source of instructions which can be used to hide latencies. However, because these architectures can execute instructions from different threads each cycle, there is no fundamental difference (from the CPU scheduler's point of view), whether latencies are covered with instructions from the same or different processes. Of course, the actual runtime of a multithreaded process likely depends on whether latencies are being filled from that same process or not.

An interesting feature of both the SMT and the MTA is that, if resources such as functional units (in the case of the SMT) or issue slots (in the case of the MTA) are unused by one program's threads, they can be used by another program's threads *possibly without adversely affecting the first program.* In general, if a program is not fully utilizing the resources of the machine, it leaves an opportunity for a contemporaneously executing program to use these left-over resources. If the O/S knows or can predict that a group of jobs may make progress without hindering each other it can increase system throughput by co-scheduling these jobs. Information about a job's resource usage, obtained statically or dynamically, could thus be used by the job scheduler to make good choices about what jobs to run together.

Symbiosis is defined as the mutually beneficial living together of two dissimilar organisms in close proximity. We adapt that term to refer to the increase in throughput that can occur when two or more applications are executed concurrently on a multithreaded computer. We define symbiosis in terms of throughput rate where throughput rate is given by

$$\mathbf{TR = (t_1 + t_2 ... + t_k)/SimultaneousExecutionTime(program_1, program_2, ..., program_k)}$$

and t_i is the execution time of $program_i$ when $program_i$ is executed alone. Intuitively, symbiosis exists if the time to execute a batch of programs all together is less than the sum of the times to execute them separately.

*Department of Computer Science and Engineering University of California at San Diego
[†]San Diego Supercomputer Center

We think of symbiosis as being potentially positive (beneficial) or negative (harmful). If the throughput rate is greater than one we have positive symbiosis. So we define symbiosis as

$$\mathbf{Symbiosis = (TR - 1)/(TM - 1)}$$

where

$$\mathbf{TM = (t_1 + t_2 ... + t_k)/MaximumOfExecutionTimes(program_1, program_2, ..., program_k)}$$

TM is a measure of how good symbiosis could possibly be (at least in any probable scenario); at best all the jobs run together in the same time as the longest of their individual runtimes. If TR = TM (that is, the longest job does not slow down at all despite coscheduling) we have positive symbiosis of 1. Zero symbiosis occurs when TR = 1 (no speedup is achieved through coscheduling), and negative symbiosis occurs when coscheduling on a multithreaded machine actually lowers overall throughput. Note that symbiosis is a characteristic, not of individual jobs, but of groups of jobs in execution. We expect symbiosis on a multithreaded machine to be between 0 and 1; however, it could be arbitrarily negative. This could be due to high rates of interference between the jobs on a particular resource, such as instruction or data caches. Symbiosis can also be degraded because of the extra overhead required for swapping between jobs.

2 Observations on the Nature of Symbiosis

Symbiosis as defined above is certainly not unique to multithreaded architectures. It has long been known that for programs that make substantial use of I/O devices or virtual memory, concurrent execution can lead to a total execution time that is less than the sum of the individual execution times of the programs. While one job is stalled waiting for an I/O request to be satisfied, another job can make use of the processor. This example of symbiosis is a result of the fact that computers have two resources, the CPU and the I/O processor, that can be independently and concurrently scheduled. However, the benefits of such *multitasking* is limited to coarse-granularity scheduling, since the context switch time is many thousands of cycles.

On multithreaded architectures, there are many more resources that can be independently and concurrently scheduled. Furthermore, for all these resources, there is no runtime overhead associated with the dynamic allocation of the resource. Zero cost context switching provided by the MTA and simultaneous existence of contexts on the same cycle in SMT mean that these machines can cover short latencies with instructions from other programs. Our study is motivated by a desire to understand how programs interact on multithreaded architectures, why they do (or do not) achieve symbiosis and how symbiosis varies during the execution of a group of programs.

If two equal-length programs run together in the same time that either would run by itself, the symbiosis is 1. It is conceivable, but hard to construct real examples such that, the symbiosis of two programs is greater than 1. A simple experiment demonstrates positive symbiosis; if a collection of programs on the Tera MTA are compiled with threading inhibited (so each uses only one stream), then a number of programs equal to 21 times the number of processors run simultaneously in the same time as one program. This is because the effective depth of the instruction pipeline is 21 cycles and each thread can have only one instruction in the pipeline at a time.

If we observe symbiosis greater than 1 it indicates that two or more programs are somehow helping each other along, perhaps by prepaging data or filling cache lines for each other on the SMT. They have to be helping each other by a factor greater than they are hurting each other by competing for other shared resources. We consider such conditions unlikely to arise in practice. In this study we have chosen not to try to deliberately concoct programs that exhibit such behavior.

We expect that symbiosis has no practical negative limit. It is easy to imagine scenarios where competition for limited resources degrades the performance of coscheduled processes. We suspect that symbiosis will be best for programs that do not use the full resources of the machine and so leave windows of opportunity for competing programs. This may imply that programs that are distinctly different (e.g. one is highly floating point intensive and the other is integer intensive or one is very memory intensive and the other is not) will be more symbiotic than programs that require similar resources.

For any architecture, there are multiple allocatable resources which can be cycle-shared (allocated per cycle to any available user of the resource). When two or more programs are running, if one does not use such a resource another can step in and use it. On the MTA and SMT many resources can be shared at the granularity of a cycle. In fact the SMT can issue to functional units on behalf of more than one program on the same cycle. Table 1 provides a subset of the cycle-shareable resources for the MTA and SMT.

Table 1: Cycle-shareable Resources

MTA	registers	fetch add cache	streams	memory network bandwidth	i-cache	instruction issue slots	TLB
SMT	fp regs	int regs	IQ entries	mem cache bandwidth fetch bandwidth	d&i-cache	functional units	TLB

Symbiosis is likely to be a function of how well program requirements compliment each other. Notice that this implies that threads from different applications are in some cases more likely to be symbiotic than threads from the same application using loop-level parallelism (where each thread uses the execution resources in the same proportion). This phenomenon has been observed in other SMT experiments [7] [8]. Knowing how much of the cycle-shared resources are being used by each program is a place to start for predicting symbiosis. In the next section we detail symbiosis predictions and experiments on both the MTA and the SMT.

3 Symbiosis Prediction and Observation

3.1 Tera MTA

As a starting place for the exploration of symbiosis we chose to run the five NAS 2.3 Parallel Benchmark [2] kernels in each of 15 possible pairings (including self pairings) on the Tera MTA at the San Diego Supercomputer Center [5] and in an SMT simulator. As has been our method in previous work [1], we started with the serial formulation of these kernels. The MTA's compiler automatically parallelizes these serial codes. For now, one must hand code thread decomposition for the SMT. However, because the SMT can efficiently use only a few threads per processor [3], meaningful experiments can be done by competing two single threaded benchmarks. We will be experimenting with multithreaded versions in the near future.

Table 2 shows dynamic execution profiles of the five NPB 2.3 kernels class size W running alone on one cpu of the MTA. Table 2 reports these profiles for untuned benchmarks. Each benchmark has an initialization section which is untimed when reporting benchmark performance and a timed *grind* section. Because the initialization sections take different times for different benchmarks, and because the initialization sections are often (but not always) serial, it is important to know the times of both sections when attempting to predict symbiosis. Serial sections on the MTA can issue only once every 21 cycles (the effective depth of the pipeline) and so provide ample opportunity for symbiosis. The MTA has hardware counters which can be used to count phantoms and memory references. A *phantom* is an issue slot that could not be used by the program due to dependencies and/or excessive operational latencies. We report, for each benchmark, the percentage of issue slots in grind that were phantom. Counting memory references on the MTA is important because memory network bandwidth is a finite cycle-shared resource. Two highly memory intensive programs may hamper each other's progress by flooding the network with traffic and increasing memory latencies. We report the percentage of issues in grind that made references to memory.

Table 2: Time and Resource Utilization Profiles of NPB 2.3 Vanilla on MTA

Benchmark	Tinit	Tgrind	Phantom %	Memref %
Conjugate Gradient	5.2s	3.45s	17%	97%
Embarrassingly Parallel	.24s	29.43s	63%	35%
Fourier Transform	3.7s	20.49s	74%	65%
Integer Sort	30.17s	.32s	53%	65%
Multigrid	1.62s	5.93s	26%	79%

Figure 1 show symbiosis for each of the 15 possible pairwise combinations of co-scheduled kernels run on one cpu of the MTA. All combinations are symbiotic. EP is highly symbiotic with all others benchmarks

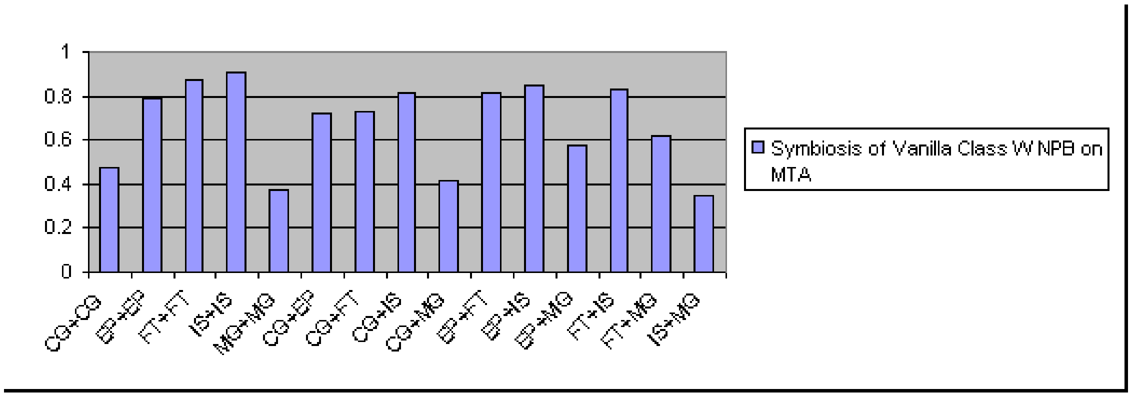

Figure 1:

and itself. This might have been predicted from its high phantom and low memory reference percentages. IS is also highly symbiotic even with other memory intensive kernels. Note that IS has a long initialization section. In the untuned version this is purely serial. Most other kernels, when co-scheduled against IS, run to completion during IS's initialization when IS is using only 1/21 of available issue slots. MG, a memory intensive program with a short initialization is less symbiotic in general. CG is highly memory intensive but has the second longest serial initialization section. This helps it achieve symbiosis with other kernels. FT has the highest phantom percentage which leaves a competing program lots of issue slots.

Table 3 gives dynamic profile information for the highest level of tuning from [1]. A primary side effect of tuning is to decrease phantom percentages.

Table 3: Time and Resource Utilization Profiles of NPB 2.3 Tuned on MTA

Benchmark	Tinit	Tgrind	Phantom %	Memref %
Conjugate Gradient	8.2s	3.23s	3%	87%
Embarrassingly Parallel	.23s	13.35	31%	18%
Fourier Transform	1.69s	2.87s	12%	75%
Integer Sort	35.89s	.26s	43%	66%
Multigrid	2.9s	3.33s	17%	85%

Examination of Figure 2 below shows that symbiosis has been reduced for all self pairings except CG+CG. Other pairings show reduced symbiosis except when paired with IS. We believe it to be a general principle (also observed in [9])that programs which are highly tuned for instruction level parallelism, by utilizing the resources of a machine more fully, present less opportunity for symbiosis with competing programs. In tuning CG we used a bit packing optimization to reduce memory references [1]. The effect of this was to lengthen the serial initialization time while decreasing grind time. This increases opportunities for symbiosis during initialization and decreases it during grind. In the case of IS, the percentage of time spent in grind has been reduced. IS still presents excellent opportunities for symbiosis during its long initialization phase.

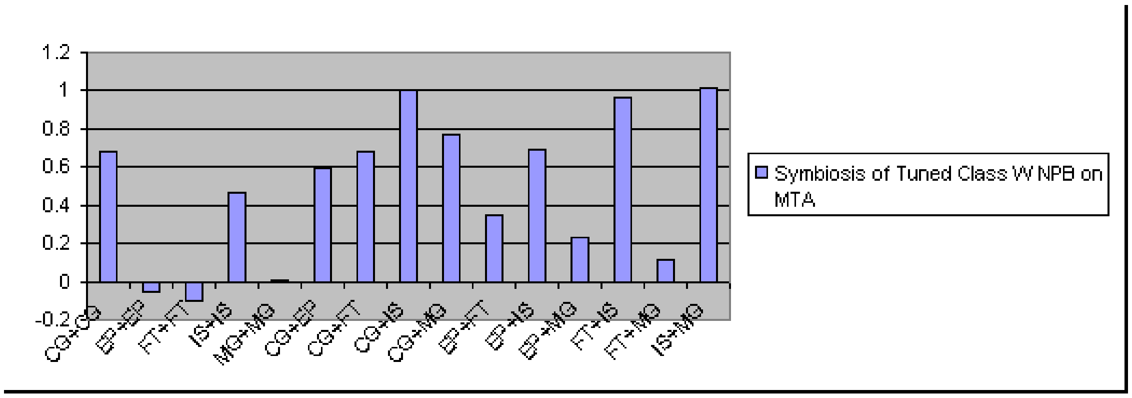

Figure 2:

Although the overlap between serial and parallel sections was not the primary effect we were looking for, many real programs do have dynamically varying degrees of parallelism. The average amount of TLP and its variability over time clearly is a factor influencing symbiosis.

To disambiguate the symbiotic effect of serial execution sections overlapping with parallel sections of another kernel from the (presumably less symbiotic) effect of two kernels running parallel at the same time, we created versions of the benchmarks which spend a much greater percentage of their time in grind. Figure 3 below shows symbiosis for tuned kernels that spend at least 90% of their time in grind and which, when run together, spend at least 40% of their time in competition with the other kernel in parallel grind. Figure 3

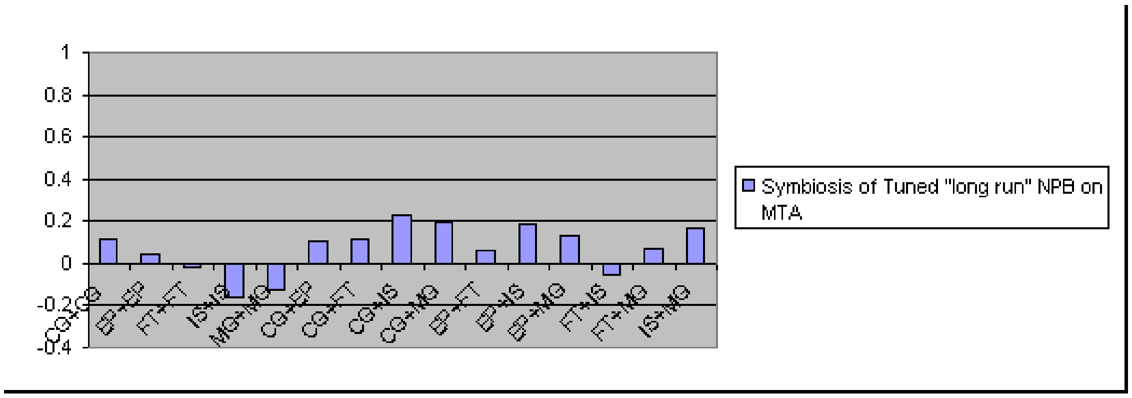

Figure 3:

confirms that these highly efficient parallel programs run on one cpu of the MTA exhibit reduced symbiosis. This is because nearly all of the available system resources are used by each program when run by itself. There is reduced opportunity for co-scheduled programs to compute symbiotically. On the other hand, symbiosis of 0.2 for two highly tuned parallel programs compiled to use the whole machine is not necessarily ”bad”. A further effect which can contribute to poor or even negative symbiosis when two parallel programs compete on the MTA is that the first program to obtain streams may cause the second program to get fewer than it requires for optimal performance. A program, during execution, asks for a sufficient number of streams to satisfy the compiler's estimate of latencies. A program needs at least 21 streams just to cover the instruction pipeline latency. Still, if fewer streams are available the program will take what it can get. A pathological case can arise when the first program to reserve streams actually only needs them for a short time. Since stream allocation is not dynamic by default, the second program will continue to be throttled by the insufficient number of streams it obtained even after more streams are actually available. This may be the case with FT+IS for example; FT finishes shortly after IS goes parallel and thus affects the resource available to IS.

3.2 Simultaneous Multithreading Processor

The SMT machine uses multiple threads to keep the functional units of a superscalar processor highly utilized. A single program or thread often does not have sufficient instruction level parallelism (ILP) to issue to all of the functional units of a superscalar processor on every cycle. The SMT uses instructions from a mix of threads in an attempt to fill more of these functional unit issue slots [7]. A single thread with high ILP does not need another thread to achieve high utilization of functional units (assuming that it also has a good mix of instruction types). One might hypothesize that threads with low ILP or threads with complimentary mixes of floating point, integer and load/store instructions would achieve symbiosis on the SMT. The SMT machine as proposed, has a data memory hierarchy so that secondary effects with respect to caching become a concern [4]. For the MTA simply characterizing the memory intensity of codes can give insight into how they compete for cycle-shared memory bandwidth. On the SMT patterns of data reuse in cache may allow positive symbiosis or lead to thrashing and negative symbiosis [3].

Our results should be interpreted differently for the SMT than for the MTA for at least two reasons. Low symbiosis on the MTA for multithreaded programs may simply signify that each is able to use the machine very efficiently by itself. Because we used single threaded programs for the SMT, a lack of positive symbiosis between two programs does indicate that multithreading is ineffective. However, unlike the MTA, a single thread on an SMT processor *can* use all of the execution resources of the processor and have negative symbiosis with another single-thread program.

Our SMT simulator processes unmodified DEC Alpha executables and uses emulation-based , instruction-level simulation to model in detail the processor pipelines, hardware support for out-of-order execution, and the entire memory hierarchy, including TLB usage [4].

We first ran the five NPB kernels on our SMT simulator with direct-mapped L1 data and instruction caches. Our experiments showed that self-pairings thrashed the L1 instruction cache. To gauge the importance of this thrashing on symbiosis, we performed the same experiments, this time with 2-way set associative L1 caches. Table 4 and Figure 4 below give the results for this configuration. This change improved the symbiosis of self-pairings in three of the five cases quite dramatically. However, for reasons we discuss below, the symbiosis in four of the five cases remained negative.

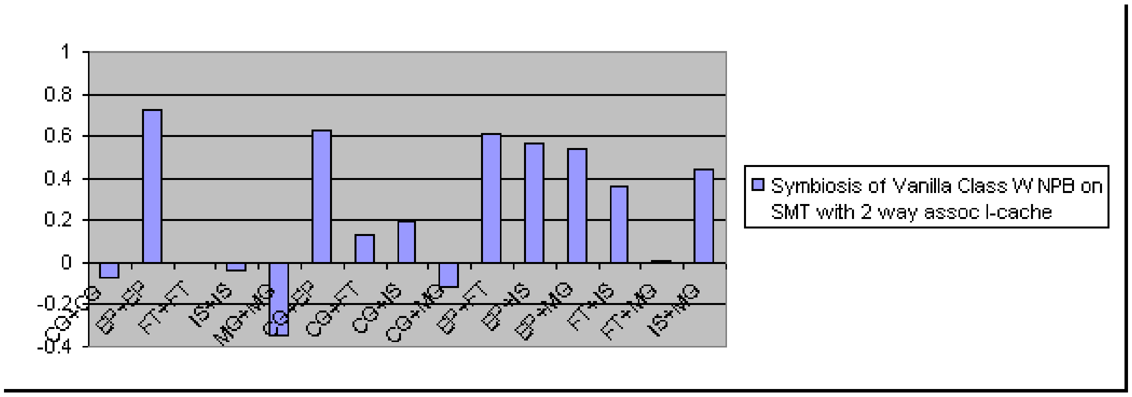

Figure 4:

All the self-pairings on the direct-mapped configuration suffer from thrashing. For example, the average memory latency when pairing CG nearly doubles, and icache hit rates drop precipitously.

When we use 2-way associative L1 caches, the CG+CG pairing speeds up by 45%. However, the CG+CG pairing remains negatively symbiotic. The problem is two-fold. First, L3 cache hit rates remain bad, at 3.95%. Secondly, now that cache hit rates have increased, a bottleneck has been exposed in the floating-point queue: in 60% of the cycles, the floating point queue was full.

Table 4: Resource Utilization Profiles of NPB 2. 3 SMT with 2 way assoc. I-cache

Benchmark	IPC	instruction mix FP INT LD/STOR	queue conflict IQ FPQ	register conflict INT FP	active-list overflow or conflict	fu conflict INT FP LD/STOR	cache miss rate I DL1 DL2 DL3	DTLB miss rate
CG	1.770	.255	.096	.014	.024	.009	8.97e-05	.06
		.342	.253	.295		.001	.219	
		.404				.18	.267	
							.131	
EP	1.011	.374	.289	9.89e-05	.003	.012	1.99e-05	.010
		.373	.503	..0004		.013	.013	
		.253				.008	.968	
							.001	
FT	2.629	.339	.334	.008	.026	.060	1.26e-05	.04
		.236	.184	.055		.183	.175	
		.425				.097	.233	
							.334	
IS	1.139	.098	.272	.238	5.01e-05	.003	1.60e-05	.17
		.428	.286	5.5e-05		7.3e-05	.175	
		.479				.005	.223	
							.571	
MG	3.063	.396	.066	.003	.001	.050	1.03e-05	.03
		.181	.103	.370		.082	.042	
		.423				.132	.630	
							.007	

IS+IS faired similarly to CG. While increasing associativity drops cache miss rates, it exposes the bottleneck of integer registers. With 2-way associative L1 caches, the processor runs out of integer registers in 32% of the cycles.

MG sees almost no gain by increasing associativity: symbiosis drops only from -0.36 to -0.35 by doubling associativity. The main bottleneck, regardless of associativity, is floating-point registers. For example, with direct mapped caches, in 60% of the cycles the processor exhausts its pool of 100 floating point registers. With 2-way associative caches, this figure increases to 65%.

EP, on the other hand, saw very good symbiosis in both machine configurations: 0.6 and 0.73 on direct mapped and 2-way configurations. This symbiosis did not come about due to high cache latencies (which would allow overlap of communication and computation). Instead, the major reason behind this result is the low instruction level parallelism in EP. In both configurations, EP averaged 1 instruction per cycle throughput, leaving many of the issue slots idle each cycle. When paired with itself, this figure increased to 1.42 and 1.51 in the two configurations.

The CG+MG pairing faired poorly in both configurations, with a symbiosis of -0.12. The problem was not associativity: increasing associativity decreased the average memory latency by only 6.5%. The bottleneck was floating-point registers. In both configurations, roughly 47% of the cycles saw an overdemand for floating point registers. By themselves, CG and MG had floating point register overdemand in 29% and 37% of the cycles.

FT+MG showed a symbiosis of near zero. The bottleneck in this case was a combination of floating point registers and floating point issue slots. In 26% and 14% of the cycles, respectively, these resources were constrained.

Interestingly, all applications, when paired with EP, show higher symbiosis than when EP is paired with itself. The major reason for this disparity is due to the much higher L3 cache hit rates in the dissimilar pairings. For example, while EP+EP had a 0.74% L3 hit rate, CG+EP showed a hit rate of 77%. A less important reason is due to the slightly more equal instruction mixes. For example, the mix of instructions on EP+EP was 37% floating point, 37% integer, 25% load/store. With CG+EP, the mix became 34%, 37%, 29%.

IS is not floating point intensive and exhibits positive symbiosis with floating point intensive codes.

4 Future Work

We have demonstrated that programs co-executed on multithreaded machines can exhibit positive symbiosis. In a multiprogrammed environment, a serendipitous choice of jobs to co-schedule could result in improved system utilization and throughput. Alternately, an unlucky choice could lead to degraded throughput. With more study, a clear picture of the factors likely to lead to symbiosis between two or more programs executed on a multithreaded machine seems likely to emerge. If static predictions or dynamic profiles can be used a priori to arrive at a symbiotic schedule for a batch of jobs, multithreaded servers may be able to achieve superlinear throughput.

A feature of our definitions of throughput and symbiosis is that one can compute these for small intervals of time, and combine them for aggregate periods using simple and familiar arithmetic. In particular, if the throughput during an interval of s1 seconds is t1, and during the next s2 seconds it is t2, then the throughput for the entire (s1+s2) seconds is the weighted average of the t's, i.e. t1(s1/(s1+s2) + t2(s2/s1+s2). This is also true of symbiosis. This property will be useful when we perform experiments in which the jobs start and complete at different times. We also wish to precisely characterize symbiosis during intervals when parallel sections of one program overlap with serial sections of another and when parallel sections overlap. This will be particularly interesting when dynamic thread allocation is done.

5 Acknowledgements

This work was supported in part by NSF award ASC-9613855 and MIP-9701708, DARPA contract DABT63-97-C-0028, UC Microgrant 97-018 and Digital Equipment Corporation external research grant US-0040-97.

References

[1] "NAS Benchmarks on the Tera MTA" J. Boisseau, L. Carter, K. Gatlin, A. Majumdar, A. Snavely, *Workshop on Multi-Threaded Execution, Architecture, and Compilers,* February 1998.

[2] See http://science.nas.nasa.gov/Software/NPB

[3] "Contention on 2nd Level Cache May Inhibit the Effectiveness of SMT" S. Hilly, A. Seznec. *Workshop on Multi-Threaded Execution, Architecture, and Compilers,* February 1998.

[4] "Tuning Compiler Optimizations for Simultaneous Multithread ing" Jack L. Lo, Susan J. Eggers, Henry M. Levy, Sujay S. Parekh (UW) and Dean Tulls en (UCSD) *Proceedings of Micro-30* , December 1997.

[5] See http://www.sdsc.edu

[6] "Multi-processor Performance on the Tera MTA" Allan Snavely and Larry Carter (SDSC/UCSD), Jay Boisseau and Amit Majumdar (SDSC), Kang Su Gatlin and Nick Mitchell (UCSD), John Feo and Brian Koblenz (Tera Computer) *Supercomputing 98,* Orlando

[7] "Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading" Jack L. Lo(UW), Susan J. Eggers(UW), Joel S. Emer(DEC), Henry M. Levy(UW), Rebecca L. Stamm(DEC) and Dean Tullsen (UCSD) *ACM Transactions on Computer Systems* , August 1997.

[8] "Simultaneous Multithreading: Maximizing On-Chip Parallelism" D.M. Tullsen, S.J. Eggers, and H.M. Levy,*22nd Annual International Symposium on Computer Architecture,* June 1995.

[9] "Optimization Issues for Fine-grained Multithreading: The impact of flexibility on complexity" Nick Mitchell, Larry Carter, Jeanne Ferrante, and Dean Tullsen, *work in progress*

View publication stats