A characterization of the SPARC T3-4 system

van Tol, M.W.

Publication date
2011

Document Version
Final published version

Link to publication

Citation for published version (APA):
van Tol, M. W. (2011). *A characterization of the SPARC T3-4 system*. (arXiv.org/cs; No. arXiv:1106.2992). Cornell University Library. http://arxiv.org/abs/1106.2992

A Characterization of the SPARC T3-4 System

Michiel W. van Tol
Computer Systems Architecture group, Institute for Informatics
Faculty of Science, University of Amsterdam
Amsterdam, The Netherlands
mwvantol@uva.nl

Abstract

This technical report covers a set of experiments on the 64-core SPARC T3-4 system, comparing it to two similar AMD and Intel systems. Key characteristics as maximum integer and floating point arithmetic throughput are measured as well as memory throughput, showing the scalability of the SPARC T3-4 system. The performance of POSIX threads primitives is characterized and compared in detail, such as thread creation and mutex synchronization. Scalability tests with a fine grained multithreaded runtime are performed, showing problems with atomic CAS operations on such physically highly parallel systems.

Contents

1. Introduction

This document is a technical report discussing tests and results on a Sun/Oracle SPARC T3-4 system that the Computer Systems Architecture (CSA) group at the University of Amsterdam had access to through the CMT beta program. The experiments described in this document were solely conducted by the author, and do not provide a full account of all tests done on the T3-4 by the CSA group within the test period. The opinions expressed in this document are solely those of the author, and are not official opinions of neither the CSA group nor the University of Amsterdam.

The T3-4 system is a 4 way multicore SPARC T3 [1] system with the CPUs running at 1.67 GHz. Each CPU contains 16 cores, and each core contains 8 hardware thread contexts, for an aggregated total of 512 hardware threads in the whole system. For a more detailed description of the architecture and organization of the system, we refer to the available literature [1, 2, 3, 4].

The rest of this document is structured as follows; first, we will discuss the background of the research the author is involved in within the CSA group, to show from which standpoint these experiments were selected. Then we will thoroughly discuss each individual experiment; how it was defined, analyze its results, and when applicable, compare the test results to two reference systems. After all experiments have been discussed, we draw our overall conclusion about the T3-4 system. The two reference systems that we used consist of a Sun/Oracle X4470 server [5] fitted with four 6-core Intel Xeon E7530 [6] processors running at 1.87 GHz, and a Dell PowerEdge R815 [7] AMD *Magny Cours* system which has four 12-core AMD Opteron 6172 [8] processors running at 2.1 GHz. Both systems are running CentOS 5.5 64-bit Linux, kernel version 2.6.18. As the Xeon E7530 processor has HyperThreading, giving us two threads per core, both reference systems present themselves through the OS as 48-cpu machines. Throughout this document we will refer to these machines as *X4470* and *Magny Cours*. The T3-4 system is running Oracle Solaris 10 9/10.

2. Background

As the development of performance using single cores has stagnated due to the power wall, a switch has been made in the last few years towards multi- and many-core architectures. In order to exploit the potential of architectures with hundreds, thousands or more cores, concurrency should be the norm and starting many parallel tasks should be cheap. This is reflected in one of the areas the CSA group is doing its research in, where we are doing work on the design of a conceptual many-core architecture; the Microgrid [9]. This CMP architecture employs fine grained data driven scheduling between many small threads of execution, a core can accommodate up to hundreds of threads and switch between them on a cycle-to-cycle basis, allowing for fine grained latency hiding. These threads, so called Microthreads [10], can be created cheaply by the core's logic in a few cycles, and can consist of anything ranging from a few instructions to an entire program. Effectively, this approach bridges the gap between the granularity of classic instruction level parallelism and thread or task level parallelism. Another key point of this approach is that by using this latency hiding approach, we can do with small simple cores with in-order pipelines and smaller caches, allowing us to potentially put more of these cores on one single chip.

The Microgrid architecture is an implementation of the SVP concurrency model [11], which allows the expression of concurrency and dependencies down to the level of loop level parallelism. One of the contributions of the author to this research direction was the development of a software run-time[1] that implements SVP on top of POSIX threads [12], so that we could test SVP programs on modern multi-core hardware. Of course thread creation is 4 to 5 orders of magnitude slower, but it did allow us to explore many aspects of SVP. Furthermore it has led to research into a system level software SVP implementation allowing finer grained scheduling with less overhead, and we are currently working on applying this to the experimental 48-core Intel SCC [13] research chip. A future direction might be to consider a similar implementation effort on a T3 system or future generation T-series, as well as other future multi- or many-core architectures.

Coming from this angle, we see a lot of parallels between our Microgrid architecture and the CMT approach of the T-series [2]. We were then first interested to investigate the boundaries of the *T3-4* system, and to see how the processor cores behave with massively parallel workloads. As besides computationally bound, tasks can either also be memory or I/O bound, we also try to measure the maximum memory bandwidth. Unfortunately we did not have sufficient I/O hardware available to do any sensible tests in the third direction. As we are dealing with a system with this many cores, the mapping of a task will have a great impact. We first investigate that in Section 3, after which we

[1] From here on referenced as *SVP-ptl*, the SVP POSIX Threads Library

measure the Integer arithmetic and Floating point arithmetic throughput in Section 4 and do a comparison with our reference machines. Then, we test the second boundary of the machines with a Memory throughput test in Section 5. The last two test parts are based on tests relevant for the SVP-ptl run-time [12], one set of tests in Section 6 tries to measure the properties of the implementation of POSIX threads (or pthreads in short) on our test machines, and one set of tests in Section 7 running actual tests using our SVP-ptl run-time.

3. The effect of Mapping

3.1. Thread mapping experiences

With as many as 512 virtual processors, it is interesting to see what the effect of mapping your threads are, and how Solaris handles this automatically. The first thing that we had to figure out was how the processor IDs that can be passed to the ***processor_bind()*** function are related to the actual processors, cores and thread slots on the *T3-4*. In our initial experiment we assumed a 'dumb' mapping, where thread 0 is bound to processor ID 0, thread 1 to processor ID 1, thread 511 to processor ID 511, thread 512 to processor ID 0 again, and so on. However, as expected, this does not give the optimal performance, so we tried to discover how these IDs are actually distributed. Using the ***psrinfo -pv*** command it turned out that IDs 0-127 are on physical processor 0, IDs 128-255 on processor 1, etc. Then, several tests with two threads on one physical processor revealed that every sequence of 8 IDs correspond to the eight thread slots of one single core. For example, virtual processor IDs 0-7 turned out to map to the eight thread slots of core 0 on physical processor 0. However, these measurements also revealed that one thread on ID 0 and a second one on one of IDs 4 to 7 also yielded a performance improvement for Integer Arithmetic. This can be explained by the fact that each pipeline has two Integer ALUs, as found in the SPARC T3 datasheet [3], which apparently are shared among a group of four threads. Using the information about mappings gathered by these short experiments lead to the definition of our *Round Robin* mapping scheme where every four threads are divided amongst the four physical processors, and then every sixteen threads on a physical processor are divided over all sixteen cores. This means that when mapping 64 threads, every physical core in the system gets assigned exactly one thread in thread slot 0. Then the next group of 64 threads that have to be mapped are divided similarly, but are mapped to the slots corresponding to the second ALU on each core; i.e. threads 0-63 are mapped to thread slot 0 on each core and threads 64-127 are mapped to thread slot 4 on each core. threads 128-255 then map to thread slots 1 and 5 respectively, and so on until 512 threads have been mapped and the system is completely full and mapping starts again from slot 0, core 0 on physical processor 0. For independent threads this should be the most optimal mapping, as the maximum of computing resources is exposed. For threads communicating through shared memory for example this is obviously not a straightforward mapping strategy. As a third mapping method we decided to run our experiment without calling the ***processor_bind()*** function, and let the Solaris scheduler decide how to map our threads.

3.2. Experiment setup

In our first experiment, we compare the impact on computational throughput using the three mapping methods described in the previous section. In order to measure this, we create *n* independent threads that each loop over a small input dataset of 128 items and perform 8 arithmetic operations using one input data and the result of the previous iteration. The 8 arithmetic operations consist of two additions, two subtractions and four multiplications. The calculation that is done is exactly the same for both the integer arithmetic and floating point arithmetic throughput tests. This loop is then executed one billion times, yielding in $8 \cdot 10^9$ integer or floating point operations per thread, excluding the loop overhead. These arithmetic operations are carefully composed so that the calculation could not be simplified and rewritten to less operations - or at least, that's what we originally thought. After compiling this code with both GCC [14] (Version 4.5.1 with optimization flags: **-O3 -mtune=niagara2 -mpcu=niagara2 -mvis**) and the Sun C Compiler (version 5.11 with optimization flags: **-m64 -fast -native**), it turned out that the code compiled by the Sun C Compiler performed an amazing 33% better than GCC. After inspecting the assembly code generated by both compilers, it turned out that the Sun C Compiler very aggressively managed to unroll the computational loop, even though it had dynamic bounds passed through a void* referenced struct through the ***pthread_create*** call. After unrolling the loop, it could combine several operations from the different iterations of the loop and yield only 6 instead of 8 operations per iteration. Therefore, in order to calculate throughput, we compensated our measurements to $6 \cdot 10^9$ operations per thread for the Sun C Compiler, and $8 \cdot 10^9$ operations for GCC compiled code. The mapping experiment

GIAOPS

# Threads	Dumb	Round Robin	Auto
2	0.68	0.74	0.75
3	0.88	1.11	1.13
4	0.99	1.48	1.51
6	1.47	2.23	2.26
8	1.94	2.97	3.01
12	2.90	4.45	4.51
16	3.86	5.92	6.02
24	5.79	8.89	9.02
32	7.73	11.87	12.02
48	11.58	17.80	18.02
64	15.46	23.71	23.87
96	23.16	35.47	35.56
128	30.73	46.45	43.64
192	46.09	63.55	64.48
256	54.92	82.90	70.75
384	85.45	95.35	91.48
512	116.21	105.07	112.87
768	91.58	99.07	108.89
1024	118.54	109.04	113.36
1536	120.80	111.70	113.93
2048	117.91	110.23	115.17

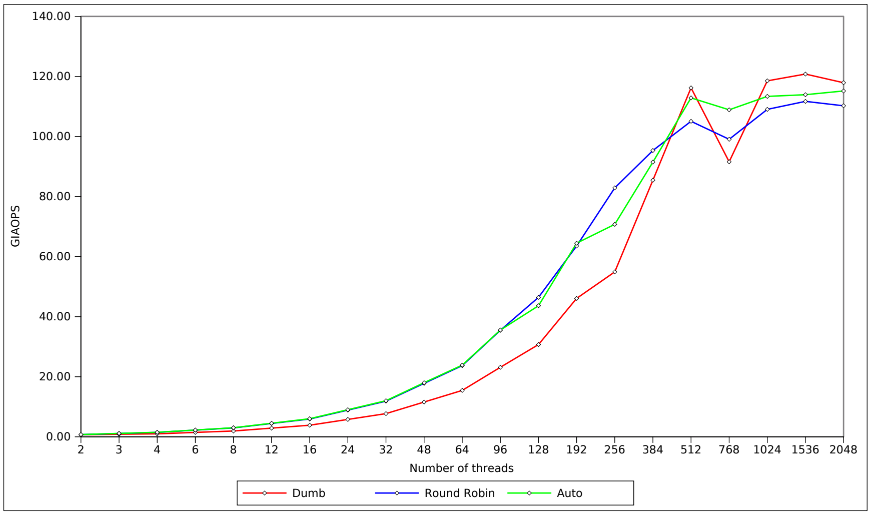

Figure 1. Integer Arithmetic throughput on *T3-4* with different mapping strategies

was carried out using the Sun C Compiler, and the comparison with GCC is further made in 4.

In order to try to start each thread's computational part at the same time, each thread first sets up the required memory, calls the ***processor_bind()*** function and acknowledges that it has started before waiting at a barrier constructed with ***pthread_cond_wait()***. When all threads have checked in, the barrier is released and each thread retrieves and stores the time from ***clock_gettime()*** using ***CLOCK_REALTIME***. Then they enter the computational loop explained earlier, and after completion call ***clock_gettime()*** again to store the time after the computation. In order to calculate our computing throughput, we iterate over the stored time values and measure the time from the first thread to start its computation, until the time at which the last one completed. Of course this is not completely accurate as not all threads will start at exactly the same time, but the computational part is proportionally large to compensate for this. Actually, there does not seem to be any better way to do this.

3.3. Results

We have run the measurements using 2^n and $2^n + 2^{n-1}$ threads using the three different mapping strategies for both integer and floating point performance. The result for Integers is shown in Figure 1 on page 5 as GIAOPS[2], we use this measure instead of, for example, MIPS as we are interested in the throughput of instructions that perform the actual computation of the kernel. As expected, the *dumb* mapping performs worse than the two other mappings due to load imbalances, until 512 threads have been reached. The dip at 768 is caused by another load imbalance; as physical processor 0 and 1 have twice the threads mapped to them than processor 2 and 3. It is surprising that the throughput for multiples of 512 threads performs slightly better than the other two mapping strategies. We have no clear explanation for this, but possibly this is a side effect of how the threads are started and synchronized with the ***pthread_cond_wait()*** barrier. As the threads most likely reach the barrier in the order they have been created, the OS probably puts them in a waiting queue in that order. As soon as the barrier is released, this is the order that the threads are woken up by the OS scheduler. When looking at the very fine grain level, only one thread is woken up at a time, so after waking up thread n there needs to be some synchronization to wake up thread $n+1$. With the *dumb* mapping this synchronization is cheap as it is almost always on the same physical processor, and even within the same core. However, with the *round robin* mapping, this synchronization is **always** to another physical processor.

We see similar behavior in Figure 2 on page 6 where we show the resulting performance for floating point calculations. The main difference with the graph on integer performance is that there is less difference between the *round robin* and *auto* mapping at 128 and 256 threads. The difference for the integer arithmetic performance is caused by the smart mapping to the two integer ALU slots done by *round robin*. As there is only one pipelined FPU, this advantage is no longer present for the floating point experiment. The rest of the graphs are similar to the integer experiment; *round*

[2] Giga-Integer Arithmetic Operations Per Second, i.e. $10^9 s^{-1}$

GFLOPS

# Threads	Dumb	Round Robin	Auto
2	0.54	0.54	0.55
3	0.81	0.81	0.82
4	1.05	1.08	1.08
6	1.46	1.62	1.64
8	1.60	2.16	2.15
12	2.39	3.23	3.29
16	3.19	4.31	4.37
24	4.79	6.46	6.55
32	6.38	8.60	8.74
48	9.56	12.92	13.10
64	12.75	17.23	17.45
96	19.13	25.84	26.01
128	25.34	34.26	34.64
192	37.99	50.68	50.65
256	45.82	58.07	55.39
384	73.32	74.45	72.82
512	94.39	84.22	87.00
768	75.77	79.34	84.08
1024	97.21	84.55	88.13
1536	97.77	87.68	92.03
2048	98.81	84.83	93.00

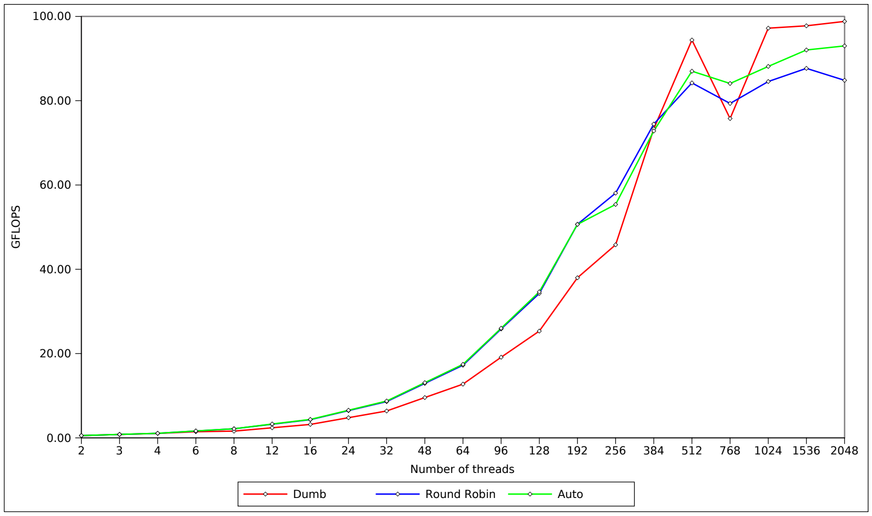

Figure 2. Floating-point Arithmetic throughput on *T3-4* with different mapping strategies

robin and *auto* mappings perform equally well, except when 512 or more threads are mapped. The *auto* mapping has the advantage here that the OS scheduler is aware of other (system) tasks outside the benchmark competing for CPU cycles, and can divide the resources in a smarter, adaptive way.

3.4. Conclusion

The main conclusion of the mapping experiments is that the *auto* mapping where the mapping is left up to the Solaris scheduler performs really well. Even though we could get a small advantage with *round robin* for certain cases, overall the OS has a better idea where and when to schedule the computations. Using this fact, we used the *auto* mapping for all our other experiments, unless other mappings were relevantly interesting to investigate.

Of course the *auto* mapping will not always perform optimal; for example, this is not necessarily the case when you have code that communicates heavily through synchronization primitives or shared memory. Then, placing your threads smartly on the system will potentially be very beneficial. Our experiences also showed that it was not easy to discover how the virtual processor IDs are laid out. We would recommend to, for example, extend the ***psrinfo -pv*** command to provide more information on how virtual processor IDs are related to actual cores. This is relevant, as we have shown the difference between mapping to thread slots and mapping to cores by the difference in performance for the *dumb* and *round robin* mappings.

As a secondary conclusion, not specific to the *T3-4*, we were really impressed by the Sun C Compiler 5.11; the fact that it very aggressively managed to unroll and simplify an all but trivial loop in our benchmark code was surprising, to say the least. Our comparison compiler, GCC 4.5.1, did not manage to apply this optimization.

4. Computing Throughput Scalability

In this experiment we try to measure the maximum computing throughput the *T3-4* can achieve, and compare it with the *X4470* and *Magny Cours* systems.

4.1. Experiment setup

The experiment setup is the same as for the mapping experiment in Section 3, using the *auto* mapping strategy. It was performed on the *T3-4* with two versions; one compiled with the Sun C Compiler and one with GCC using the versions and flags mentioned in the previous section, and using the same compensation for the number of arithmetic instructions. We then compiled the code for the *X4470* using GCC 4.1.2 with **-mtune=core2 -march=core2 -mfpmath=sse,387 -msse -msse2 -msse3 -msse4a**, and for the *Magny Cours* using also GCC 4.1.2, with **-mtune=amdfam10 -march=amdfam10 -mfpmath=sse,387**. We did not apply mapping techniques to the two

GIAOPS

	T3-4	T3-4	X4470	Magny Cours
# Threads	Sun CC 5.11	GCC 4.5.1	GCC 4.1.2	GCC 4.1.2
1	0.37	0.35	2.06	1.84
2	0.75	0.69	4.11	3.68
3	1.13	1.04	6.14	5.32
4	1.51	1.39	8.20	6.99
6	2.26	2.08	12.25	9.85
8	3.01	2.77	16.29	13.19
12	4.51	4.16	23.34	17.86
16	6.02	5.54	31.10	21.90
24	9.02	8.31	46.27	30.95
32	12.02	11.07	43.66	39.22
48	18.02	16.58	47.72	39.97
64	23.87	22.07	50.34	48.33
96	35.56	32.88	51.63	51.18
128	43.64	40.75	53.09	54.80
192	64.48	56.43	52.80	61.75
256	70.75	62.32	53.27	62.84
384	91.48	78.51	53.52	66.65
512	112.87	90.90	53.67	67.21
768	108.89	88.74	53.72	69.48
1024	113.36	91.64		70.08
1536	113.93	92.38		70.49
2048	115.17	94.18		

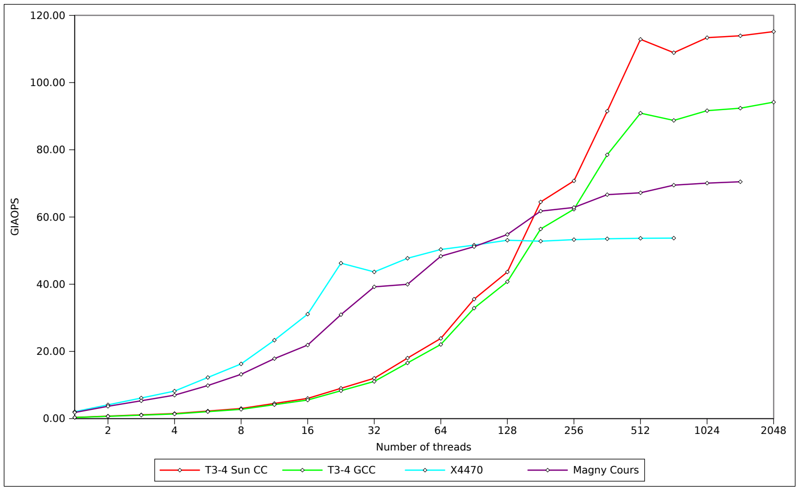

Figure 3. Integer Arithmetic Throughput Scalability

GFLOPS

	T3-4	T3-4	X4470	Magny Cours
# Threads	Sun CC 5.11	GCC 4.5.1	GCC 4.1.2	GCC 4.1.2
1	0.27	0.28	0.88	0.83
2	0.55	0.56	1.76	1.67
3	0.82	0.84	2.63	2.46
4	1.08	1.12	3.50	3.25
6	1.64	1.68	5.25	4.87
8	2.15	2.24	6.75	6.40
12	3.29	3.36	9.60	8.99
16	4.37	4.48	13.31	11.71
24	6.55	6.72	19.88	17.26
32	8.74	8.96	26.45	19.85
48	13.10	13.42	39.49	19.70
64	17.45	17.87	35.02	24.09
96	26.01	26.72	36.91	25.74
128	34.64	35.36	38.43	29.80
192	50.65	52.59	38.67	30.55
256	55.39	61.82	39.15	31.68
384	72.82	74.02	39.19	32.63
512	87.00	86.25	39.59	32.83
768	84.08	83.85	39.70	33.45
1024	88.13	87.19		33.68
1536	92.03	87.75		33.92
2048	93.00	90.44		

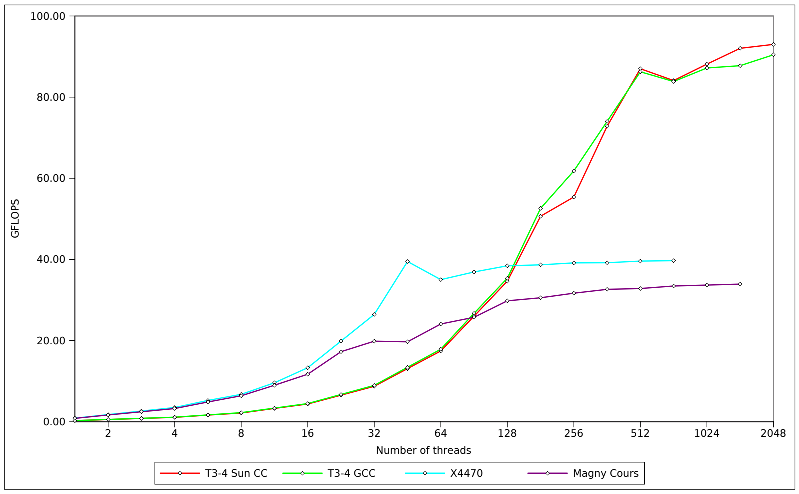

Figure 4. Floating-point Arithmetic Throughput Scalability

reference machines, so we leave this up to the Linux scheduler. As we were interested in the maximum throughput we could achieve, each measurement was carried out three times and the maximum value was used for the result.

4.2. Results

The result of measuring integer arithmetic throughput on all the three systems is shown in Figure 3 on page 7. The first thing that stands out is that the code generated by the Sun C Compiler has a much higher integer arithmetic throughput on the *T3-4* than the code generated by GCC. This is related to the loop optimization discussed earlier, as the GCC generated code has much more instructions in between to handle the loop, which compete for integer ALU resources in the pipeline. This is confirmed by looking at the floating point throughput in Figure 4 on page 7, where we do not observe such a large difference. A second thing that we can observe about the *T3-4*s performance is that it really scales up to 512 threads, requiring all thread slots on all cores to be in use to maximally utilize the ALUs and FPUs in the system.

Comparing to the *X4470* and *Magny Cours*, it is no surprise that these surpass the *T3-4* in single thread arithmetic throughput, as they have highly complex super scalar out of order execution pipelines, as well as a higher clockrate. The Xeon cores of the *X4470* outperform the Opterons in the *Magny Cours*, even though they run at a slightly slower clock. The scalability of the *X4470* is as expected, the throughput gradually increases until all 24 cores are filled up, and it gets a small boost at 48 threads due to its Hyperthreading after which it flattens out. This is even more the case for the floating point throughput, where the Hyperthreading really helps to get the maximum out of the FPUs. The scaling of the *Magny Cours* system is a bit strange, as it still keeps scaling after 48 threads, almost doubling the throughput eventually at 1024 threads. We have the impression that this could be either caused by the Linux scheduler having difficulty distributing work over such a high number of processors, or because of the input dataset for all threads which is stored in one contiguous block of memory. We observe the same effect for both integer and floating point throughput.

An interesting note can be made on comparing both integer and floating point throughput between the *T3-4* and the two reference systems. When we look at the throughput of a single thread, not surprisingly, is outperformed by the two other systems due to its more simple in order pipeline. However, when we compare the throughput of a single core, i.e. in the *T3-4* filled with 8 threads, we have to look again at the result table for the *dumb* mapping in Figure 1 on page 5 and Figure 2 on page 6. Here we see that one single core of the *T3-4* has a throughput of 1.98 GIAOPS, which is very comparable to the 2.06 and 1.84 of the reference systems, but even surpasses them with 1.60 GFLOPS against 0.88 and 0.83 respectively per core. The 1.60 GFLOPS per core show that the fully pipelined FPUs can really accept an operation every cycle as the cores clock at 1.65 GHz. Actually this last comparison did not take the Hyperthreading into account on the *X4470* as we did not measure this separately, but judging from the scaling in Figure 4 on page 7 this would come down to roughly 1.7 GFLOPS. Also we would like to note that it is unlikely that the any of the systems could use vector instructions (i.e. MMX/SSE or VIS) to speed up the calculation, and a separate benchmark would have to be constructed to compare the performance of these extensions.

4.3. Conclusion

It really takes 8 threads per core on the *T3-4* to saturate all its execution units, and the system really scales up to 512 thread workloads computationally wise. The performance of a single core (not a single thread) is very comparable to that of the *X4470* and *Magny Cours* system, given it has enough threads to execute. This exposes the nice property of how the *T3-4*s cores do latency hiding between threads, very similar to the Microgrid architecture designed by the CSA group, as discussed in Section 2. Even though we did not test the vector instruction extensions on any of the test systems, we can still conclude that the *T3-4* is likely to deliver a large computational throughput for embarrassingly parallel scientific computing applications.

5. Memory Throughput Scalability

After we explored the computational bounds of the three systems in the last section, we will now turn to the effect of memory throughput. As all three systems are 4-socket based with 4-channel DDR3-SDRAM memory interfaces per socket, this should be an interesting comparison.

5.1. Experiment Setup

The test is set up similarly as the previous two experiments, in which all threads were released (as good as) simultaneously with a barrier, and the time is measured between the first starting thread and the last thread to complete. Before the threads enter the barrier, but after (if applicable) they have assigned themselves to a virtual processor, each allocates a memory block of 256 MB. Besides allocating it with ***malloc()***, each thread also writes to the complete block with ***memset()***, to prevent the operating systems from “optimizing” the allocation using pagefault handlers and only actually allocating once a page is accessed. For the writing experiment, after the barrier and reading out the timer, the thread writes zeros to the entire 256 MB block using a ***bzero()*** call. We picked the memory block sufficiently large to have one long consecutive write that would be large enough for a clean measurement. Also not repeatedly reading/writing to the same memory ranges avoids any possible caching behavior as we really wanted to measure the throughput that we can obtain from the off-chip memory. For the read experiment we treat the 256 MB block as an array of integers (that has been zeroed before the barrier), loading each element from memory by adding it to a register within a loop.

As each physical processor has four memory channels associated, we again investigate the influence of mapping threads with different strategies on the *T3-4*. Similarly as in Section 3, we use the *dumb*, *round robin*, and *auto*

GB/s Read

	T3-4	T3-4	T3-4	X4470	Magny Cours
# Threads	Dumb	Round Robin	Auto	Auto	Auto
1	0.65	0.65	0.68	2.88	3.53
2	1.25	1.29	1.36	5.72	7.03
3	1.82	1.95	2.04	8.45	10.53
4	2.38	2.60	2.72	10.39	14.05
6	3.54	3.82	4.07	12.19	19.43
8	4.66	5.10	5.43	20.99	14.86
12	6.92	7.65	8.14	30.80	22.77
16	9.23	10.19	10.85	41.17	27.71
24	13.71	15.25	16.22	39.40	33.04
32	17.97	20.32	21.63	45.85	42.66
48	25.52	30.40	31.76	44.33	46.39
64	29.13	40.13	42.10	49.63	45.90
96	29.49	59.28	61.61	47.82	44.33
128	29.90	75.00	76.69	48.03	42.83
192	44.31	101.99	103.37	38.89	35.49
256	58.59	107.57	108.14		33.00
384	80.04	107.56	104.22		32.46
512	93.36	105.21	107.52		

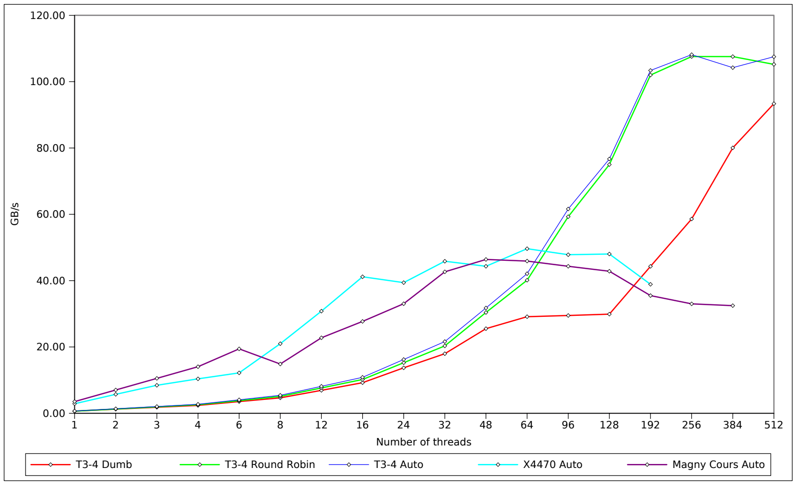

Figure 5. Memory Read Throughput

GB/s Written

	T3-4	T3-4	T3-4	X4470	Magny Cours
# Threads	Dumb	Round Robin	Auto	Auto	Auto
1	1.34	1.34	1.44	3.13	6.29
2	2.66	2.68	2.85	6.16	11.59
3	3.90	4.03	4.28	8.27	16.85
4	4.55	5.35	5.70	11.05	22.77
6	5.65	8.00	8.53	12.54	33.49
8	6.19	10.55	11.35	23.58	18.95
12	9.34	15.84	16.88	23.59	26.69
16	12.30	21.00	22.14	29.76	44.49
24	14.02	30.45	32.43	32.90	43.08
32	14.19	39.76	41.87	34.17	43.08
48	14.32	54.68	57.06	32.02	46.96
64	14.41	54.87	54.25	32.15	44.28
96	14.48	55.24	55.08	28.95	46.07
128	14.51	56.22	53.63	27.42	47.06
192	21.73	55.85	55.54	28.71	45.87
256	26.86	56.87	54.32		46.64
384	43.16	57.24	56.12		46.40
512	57.57	57.29	56.44		

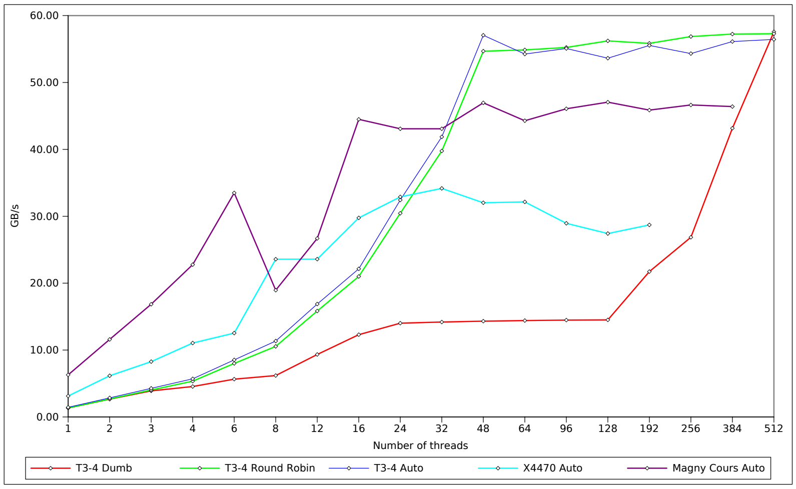

Figure 6. Memory Write Throughput

mapping schemes again. We do not specify a mapping on the reference machines, leaving up to the Linux scheduler where to map the threads.

5.2. Results

The results for reading throughput are shown in Figure 5 on page 9 and for writing throughput in Figure 6 on page 9. From the *dumb* mapping in the reading throughput graphs we can see from 64 to 128 threads that we saturate the throughput which one physical processor can achieve at 29 GB/s. And for the *round robin* and *auto* mapping there is little difference once again, and at 256 threads they manage to saturate the bandwidth of all four physical processors, coming to an aggregate of 108 GB/s. As for writing, the *T3-4* saturates a single physical processor using the *dumb* mapping at 24 threads with 14 GB/s throughput, and all four at 57 GB/s using 48 threads. The reason that the achieved throughput is lower for writing than for reading can be explained in two ways. When reading large blocks from memory, prefetching techniques will pay off, cache lines are filled, and after the first read the next few will hit in the data cache. Secondly, for memory writes, the cache line is probably fetched first on an initial write miss and then at some point written back to memory again effectively having to use the memory bandwidth twice, and potentially, snooping messages have to be sent out to every other physical processor on every write.

The results for the *X4470* and *Magny Cours* machines are also shown, and again, for single or few threads they outperform the *T3-4* system in achieved memory bandwidth. For writing, the *Magny Cours* has a big drop in throughput around 8 and 12 threads, but this might be caused by Linux not mapping the threads efficiently. At 48 threads the *Magny Cours* unsurprisingly reaches the peak of its throughput performance, but remarkably it has a better writing than reading throughput; something we repeatedly measured, and that we currently can not explain. Possibly a side effect of the coherency scheme they use. The *X4470* performs as expected, quite similar in reading behavior as the *Magny Cours*, with already at 16 threads almost saturating memory read and write throughput. It is interesting that for both reading and writing, a jump in performance is made when going from 6 to 8 threads. It could be the case that Linux by default maps the 6 threads to one physical processor as it has 6 cores, and that at 8 threads we see a larger increase in performance as threads are moved to other physical processors and use the independent memory controller there.

5.3. Conclusion

The *T3-4*s memory bandwidth scales up well for a high number of threads, and above the point that it saturates, it does not suffer much penalty. It delivers twice the throughput for reading than our reference systems. This throughput combined with the dataflow like scheduling of the threads in the cores provides enough bandwidth to accommodate many threads which due to latency tolerate can provide a large computing throughput as seen in Section 4. This property is what the CSA group also achieves in their Microgrid architecture, as discussed in Section 2, and mentioned before in the conclusion of Section 4.

6. Experiments with POSIX threads

As we now have identified the throughput bounds of the computational and memory side in the previous sections, we now will look at how the implementation of pthreads behaves on the three tested systems. We normalize each measured interval using the clockspeed of each system, in order to come to an estimated number of cycles that the measured mechanism takes. First we will look into the costs of thread creation, then thread synchronization with mutexes, and finally thread synchronization with conditionals.

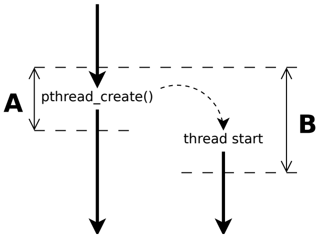

Figure 7. Measured intervals for thread creation

6.1. Thread Creation

In our first experiment with pthreads we will look at the overhead for starting threads. When using pthreads, a thread can be created in two modes; joinable or detached. Threads that are joinable allow another thread that has their identifier (usually the parent) to wait for its termination to receive back a return code, while detached threads will simply disappear on termination. For these experiments we have measured the times for the ***pthread_create()*** call to return, and the time from the ***pthread_create()*** call until the created thread starts executing its thread function. A diagram representing these two times is shown in Figure 7 on page 10.

We have measured the overhead for thread creation using 10^5 samples on all our three systems with joinable threads, and the results are shown in Figure 8 on page 11. The figure at the left-top is a scatter plot showing the relation between time *A* and time *B* as shown in Figure 7 on page 10, overlaying the results for all three machines. In this plot also a line $y = x$ is added, as all points below this line indicate situations in which the created thread started before the ***pthread_create()*** call returned in the parent. As this plot does only show how the measurements are spread, but not show their intensity, the three plots at the bottom and two histograms on the side are generated. The three plots at the bottom show an intensity plot for the dataset of each system, where we can see clearly where the common case is located. The histograms give an impression how the values *A* and *B* are overall spread, regardless of their relation.

The first thing we notice from the data presented in Figure 8 on page 11 is that the Intel Xeon based *X4470* system has the quickest thread creation, with 80% of the threads starting within 38-44 KCycles after the create call. This is much more widespread on the AMD Opteron based *Magny Cours* system where 72% of the threads are started within 58-70 KCycles. However, if we look at the *T3-4*, 70% of the threads are started in 138-146 KCycles. It is not a

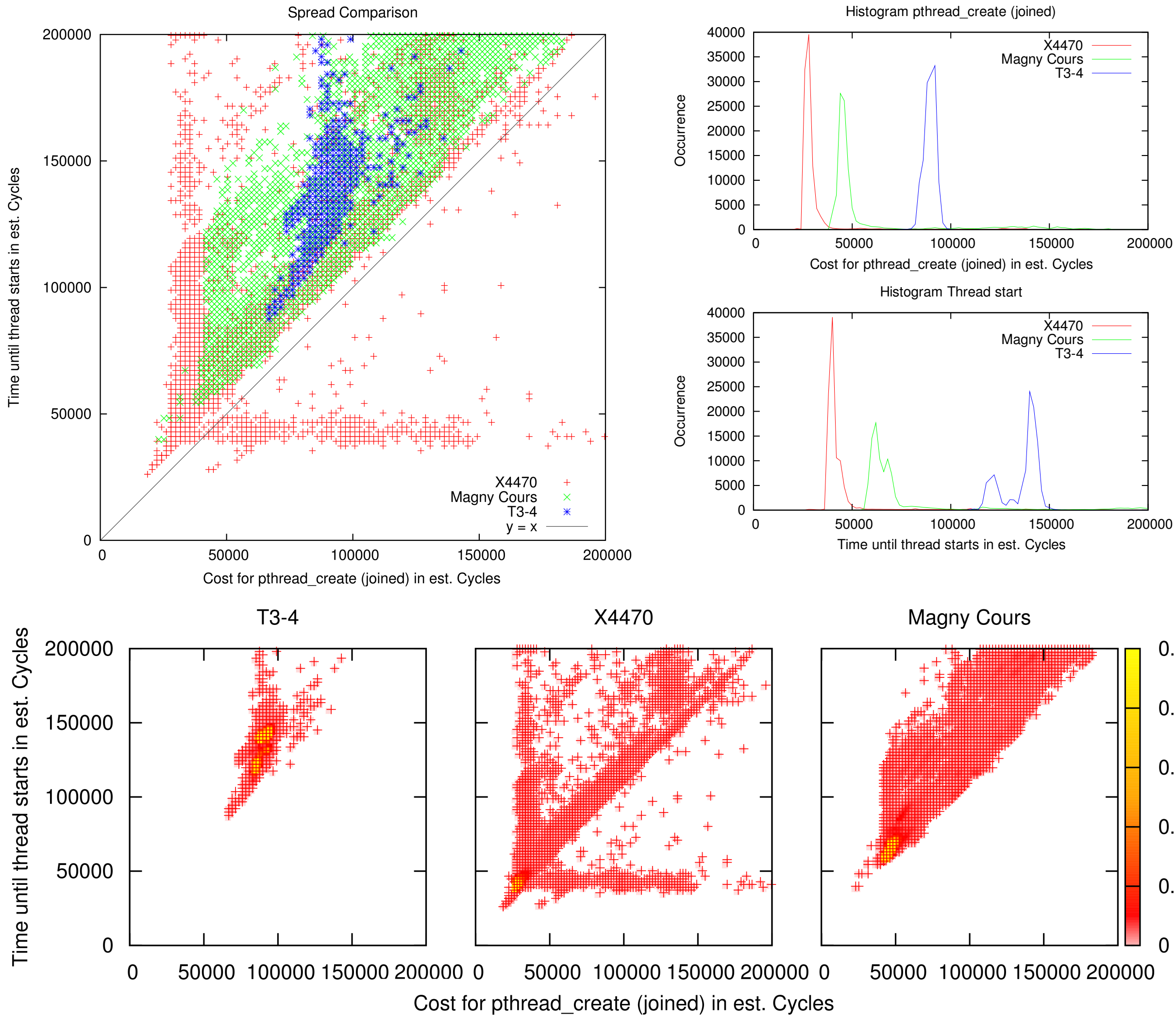

Figure 8. Measured behavior of *pthread_create()* with joinable threads

surprise that thread creation is slow on the *T3-4*, but this is not specific to the *T3-4,* as we already observed this on an UltraSPARC machine running Solaris 8 some time ago in [12], and also on other tests on a single CPU UltraSPARC running Solaris 10. Judging from the positioning of the points representing the *T3-4* in the plot, it seems that most of these cycles are spent in the parent thread, as it takes equally long to execute the ***pthread_create()*** call. This is apparently not always the case for the *X4470* system, so probably the scheduler decides to place the newly created thread on the same core, switches context, and only completes the ***pthread_create()*** call later on. It is interesting that this behavior is not seen on the *Magny Cours* system, as both systems are running the Linux 2.6.18 kernel. Probably the scheduler makes different decisions depending on the architecture of the system, or perhaps it exploits Hyperthreading by creating the new thread in the second hardware thread slot on the *X4470*.

We have repeated the same experiment as discussed above, but now for threads that are created in detached mode. The results of these measurements are presented in a similar way, in Figure 9 on page 12. The first thing we notice is that the cost for creating a thread has reduced considerably on the Linux systems. Furthermore, the *Magny Cours* platform now also shows threads that start before the ***pthread_create()*** call returned, and on average the call returns earlier. For Solaris on the *T3-4*, the creation of threads is still as expensive, but there is more variation in the cost. On the *X4470*, 74% of the threads take 32-44 KCycles to start, 69% of the threads in 44-58 KCycles on the *Magny Cours*,

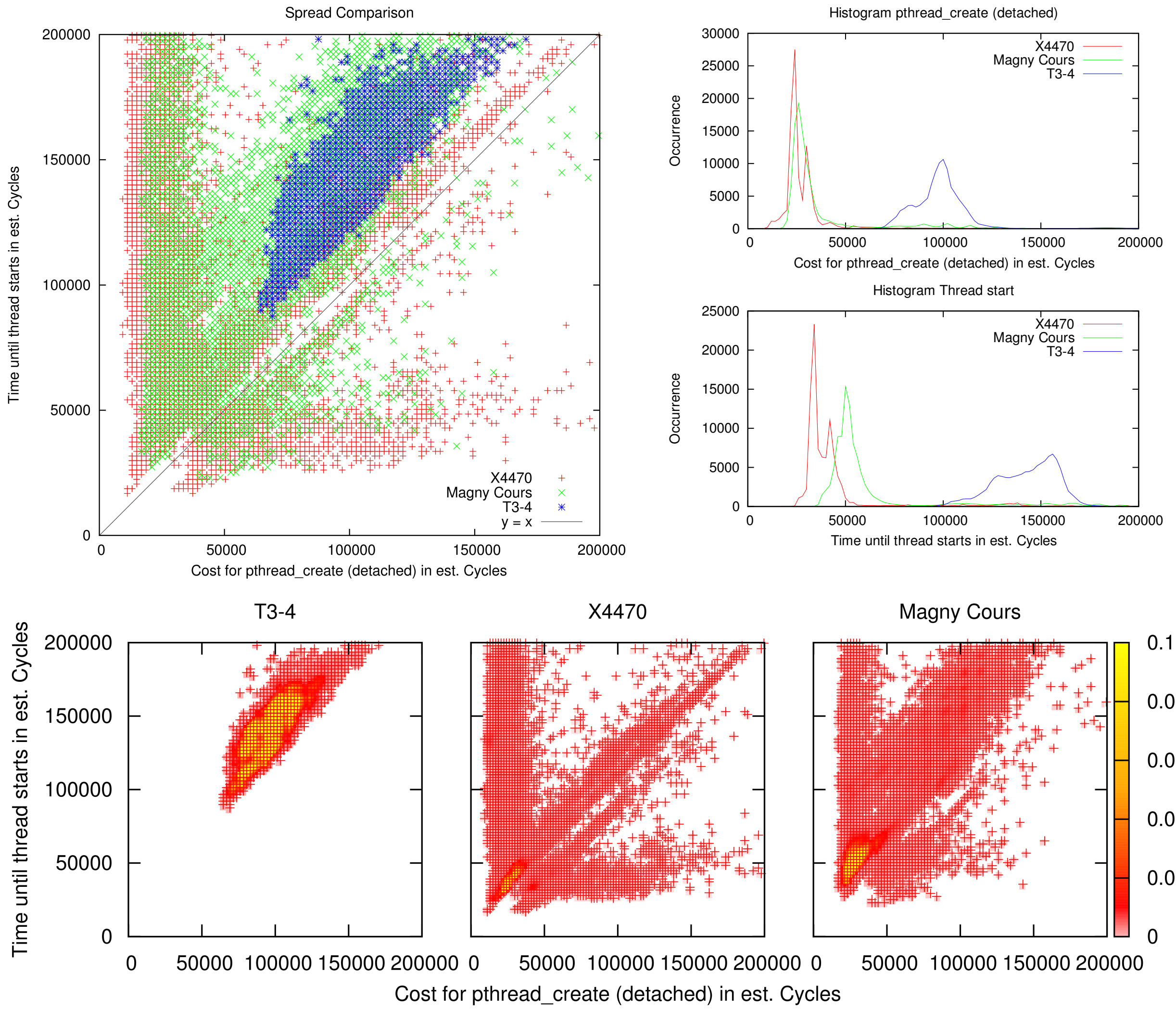

Figure 9. Measured behavior of *pthread_create()* with detached threads

and 77% of the threads in 124-160 KCycles on the *T3-4*.

6.2. Mutex Synchronization

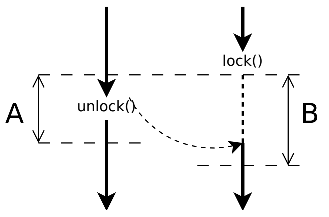

Figure 10. Measured intervals for contended mutex

Most modern operating systems implement an optimization to handle non-contended mutexes fully in userspace, to avoid the cost of making a system call and context switch to kernel mode. Linux for example implements this with a *futex*, a fast userspace mutex based on atomic operations provided in the instruction set. We measured the overhead of locking and unlocking a mutex a thousand times by a single thread, and calculated the average cost of the combination of these two operations. On the *T3-4*, this takes 432 cycles most of the time, against 46 cycles on the *X4470*, and 56 cycles on the *Magny Cours* test system. We then developed a test to see how quick a thread wakes up that was waiting on a mutex. We measure the time from the start of the unlock operation until the thread that was waiting was scheduled, as shown in Figure 10 on page 12.

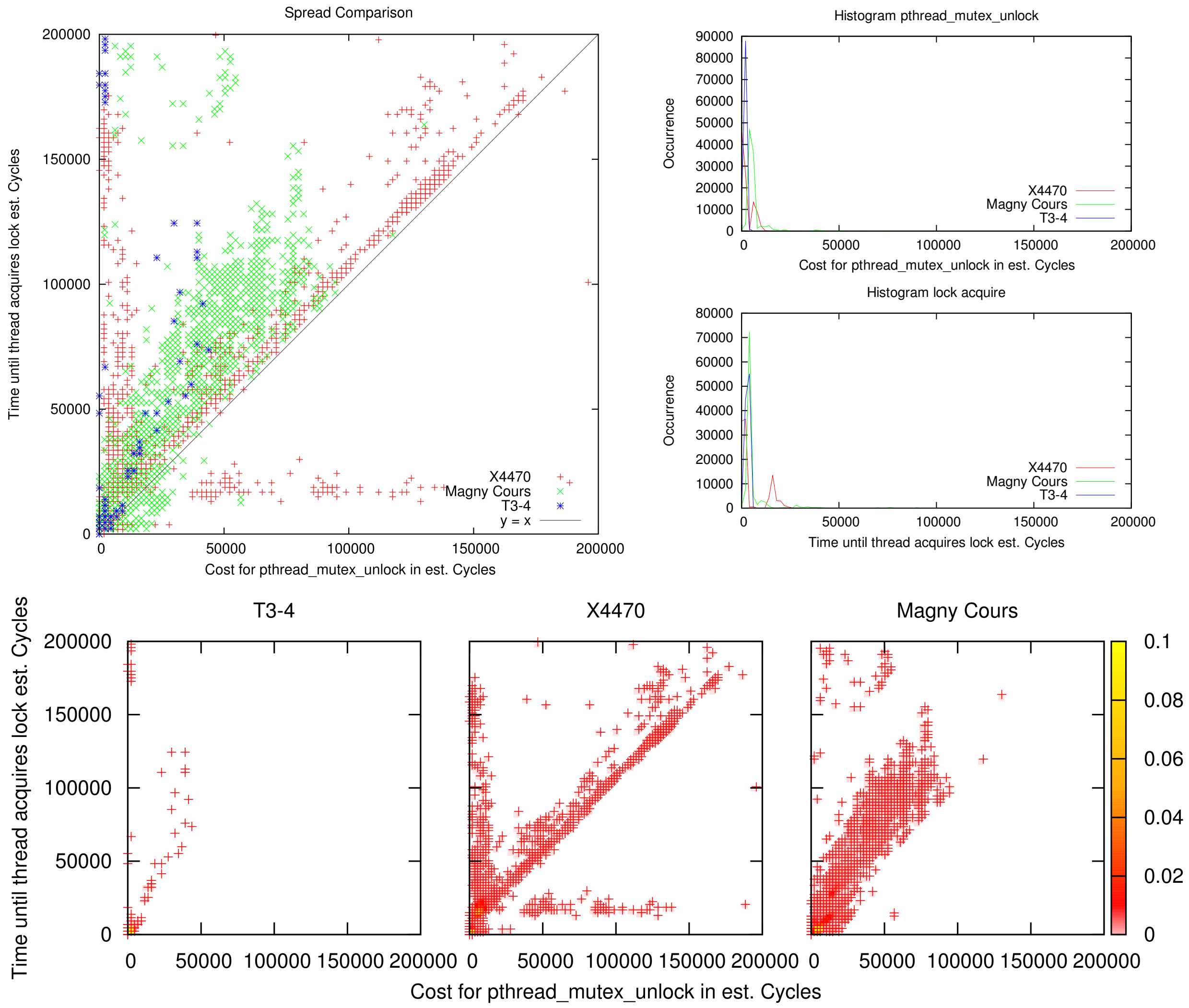

Figure 11. Measured behavior of *pthread_mutex_unlock()*

The results of 10^5 synchronizations between two threads through a mutex are shown in Figure 11 on page 13, with the same style of diagrams as in the previous section. There was no explicit mapping for these two threads, so their (relative) placement is unknown. For the *T3-4*, 99% of the threads start in less than 4 KCycles after the mutex is released, while the unlock call takes around 2-4 KCycles 99% of the cases. These figures are less concentrated for the *X4470* and the *Magny Cours*; for the *X4470*, only 73% of the threads started in less than 4 KCycles, (95% in less then 10 KCycles), and 72% of the calls to unlock took around 2 KCycles, and 90% in only less than 18 KCycles. For the *Magny Cours* system, 86% of the threads are started in 4-6 KCycles, 80% of the unlocks took 2-4 KCycles, and 92% of the unlocks were done in less than 14 KCycles.

6.3. Conditional Synchronization

A second common way of synchronization between two or more threads is through a conditional and a value guarded by a mutex. A conditional can be signaled in two ways; a single signal which will only wake up one thread that is suspended on the conditional, or a broadcast that will wake up all threads. We measure the latency of both methods between two threads, with the times we measured depicted in Figure 12 on page 14.

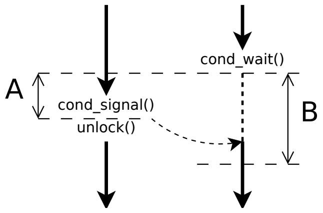

Figure 12. Measured intervals for conditional synchronization

The latency of sending a ***pthread_cond_signal()*** until the target thread wakes up are shown in Figure 13 on page 15 for 10^5 measurements, again using the similar visualizations of spread, intensity and relation. It is clear that the *T3-4* system has a predictable time in signaling the conditional with 99.9% of the calls taking around 2-6 KCycles, while on the *X4470* 83% of the calls completes after 4-8 KCycles and on the *Magny Cours*, 18% of the calls completes around 4 KCycles, but another 62% takes around 12-18 KCycles. It is interesting to see how the signaled threads respond to this in relation, as this is quite spread for the *T3-4*, on which the wake-up times are spread around three peaks; 18% is around the 42-62 KCycles area, 25% around 86-94 KCycles, and then 49% around 110-124 KCycles. The *X4470* is quite consistent as 86% of the threads wake up after 14-22 KCycles. On the *Magny Cours* we also observe something interesting; 10.5% of the threads wake up after 4-8 KCycles, corresponding with a part of the 18% of the signal calls that completed on that system in 4 KCycles, the second peak of 73% of the threads wake up at 24-36 KCycles.

We now look at the results of a similar measurement but using the ***pthread_cond_broadcast()*** call. Again, the results for 10^5 measurements are shown in the same way, in Figure 14 on page 16. At a first glance, the results are almost the same as those for ***pthread_cond_signal()***, which is not strange as we only have a single thread waiting on the conditional. Again, on the *T3-4* 99.9% of the calls take 2-6 KCycles, however it seems slightly faster on average; 4.7% at 2 KCycles versus 0.5% for the normal signaling, and 9% versus 15% at 6 KCycles. There seems to be no significant difference between the two different calls on the *X4470*, and there is a slight shift on the *Magny Cours*; as now 36% of the calls completed within 2-4 KCycles (18% previously), however, less threads started within 4-8 KCycles, only 5.9% versus 10.5%.

6.4. Conclusions

Thread creation under Solaris (on SPARC), and therefore also on the *T3-4*, is still expensive compared to the x86 based Linux systems. However, mutex synchronization is well optimized and has a very predictable behavior and outperforms the Linux mutexes roughly in 20% to 30% of the cases. The overhead for signaling or broadcasting a conditional is also low and predictable on the *T3-4*, outperforming the *X4470* in 20% of the cases and the *Magny Cours* in 80%. The downside is the longer delay at which the threads are woken up; this is mainly (99%) spread between 42 and 124 KCycles on the *T3-4*, which is worse than approximately 93% of the cases on the *X4470*, and worse than 84% on the *Magny Cours*.

In order to fully exploit the real hardware thread level parallelism offered by the *T3-4* system, it would benefit from more optimized thread creation and synchronization primitives. When such constructs cost less overhead, it becomes worthwhile to create more threads of a smaller granularity. For example, a data-parallel operation could be split into more smaller threads, able to fill up all the hardware slots in the *T3-4*. Furthermore, more functional parallelism could potentially be exploited when the overhead to create it is reduced. It would also be an advantage to

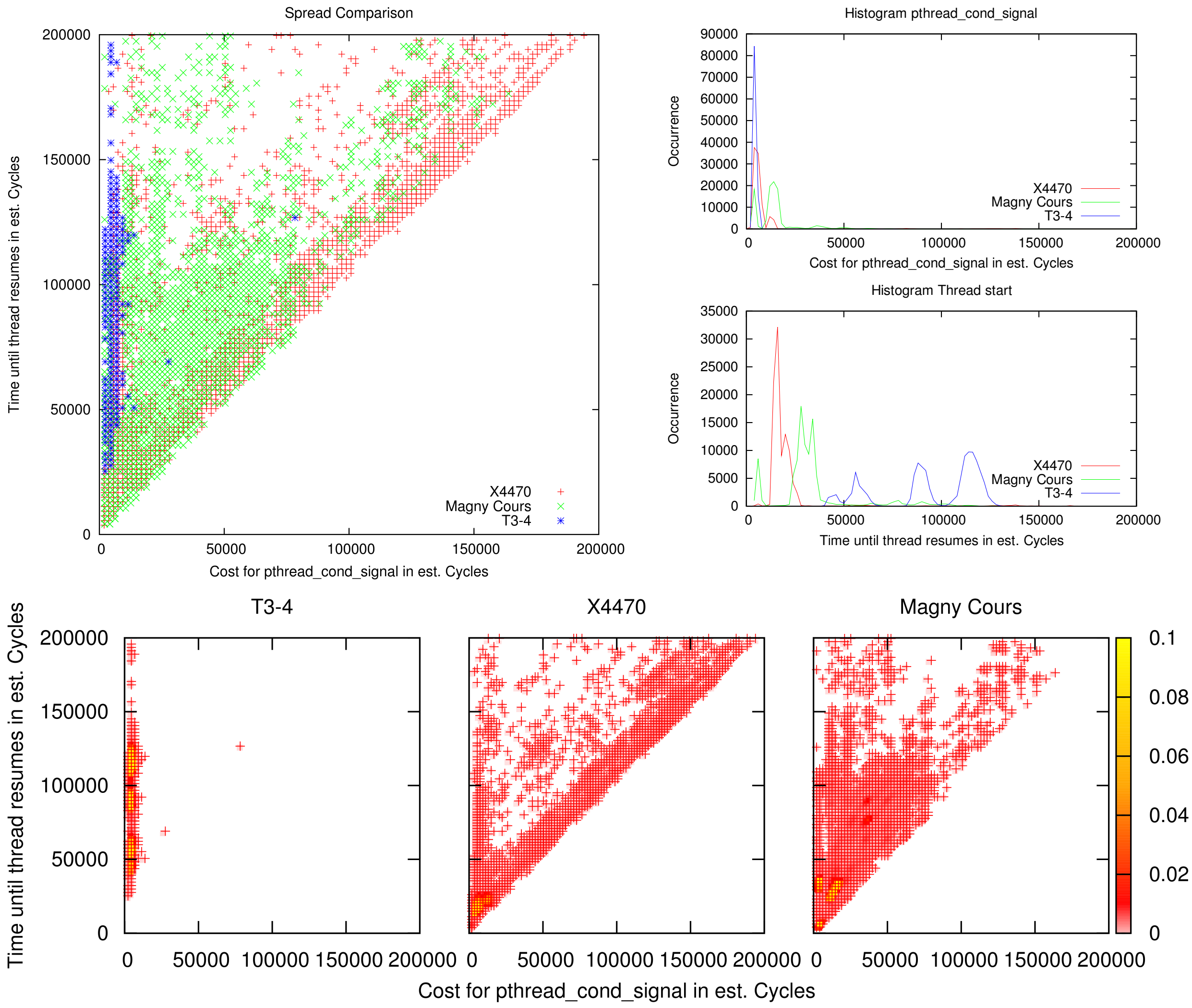

Figure 13. Measured behavior of *pthread_cond_signal()*

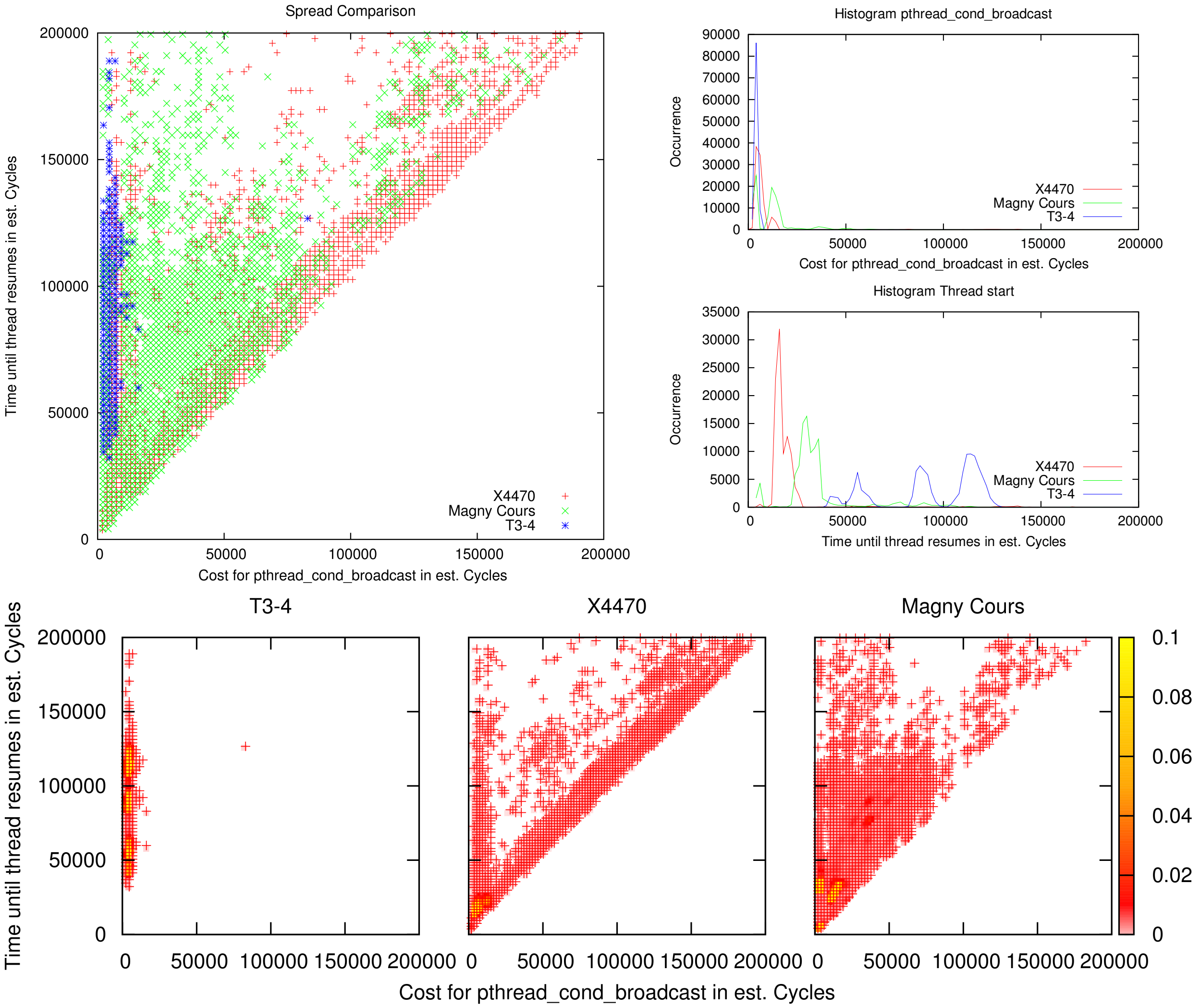

Figure 14. Measured behavior of *pthread_cond_broadcast()*

have a scheduling mechanism in user-space that can schedule work to threads mapped to the cores, for example like Apple's Grand Central Dispatch [15].

7. SVP-ptl related experiments

After determining the properties of the pthread implementations on the different systems, we now experiment with the behavior of our SVP implementation based on pthreads, SVP-ptl [12]. As there were some issues with special definitions that the Sun C compiler did not accept, the code was compiled with GCC [14] on all three machines. On the *T3-4* this was using GCC 4.5.1 with optimization flags **-O2 -m64 -mtune=niagara2 -mcpu=niagara2 -mvis**, on the *X4470* and *Magny Cours* we used GCC 4.1.2 with the optimization flags **-O2 -mfpmath=sse,387 -mtune=core2 -mcpu=core2 -mmmx -msse -msse2 -msse3 -msse4a** for the *X4470*, and **-O2 -mfpmath=sse,387 -mtune=amdfam10 -march=amdfam10** for the *Magny Cours* system.

7.1. FFT

In this experiment we run an SVP version of the Fast Fourier Transform algorithm, which was transformed to expose maximum concurrency between its iterations for execution on the Microgrid, on which it shows a nice scalability and speedup. The performance figures in this experiment do not reflect in any way the performance of the FFT algorithm on the tested systems, but they do show how the SVP-ptl run-time copes with many small short-lived threads. As creating pthreads is expensive, as also shown in Section 6, the run-time keeps a pool of pthreads that it assigns work to. When an SVP thread needs to be created, the runtime first checks the pool for an available thread, and otherwise starts another pthread. When a pthread finishes its work, it puts itself in the pool, unless it exceeded the maximum pool size. We investigate the effect of two different implementations of this thread pool here. One uses a pthread mutex to guard the pool so that it can be exclusively updated, and the second is a lockless implementation using an atomic Compare-And-Swap (CAS) instruction to add/remove entries to the pool.

For the experiment, we run the FFT implementation with the two different pool implementations for various FFT sizes varying from 1 to 16, with the number of created threads growing exponentially for each size. The results of these experiments are shown in Figure 15 on page 18, showing the execution time in ms for both implementations, and the total number of times the run-time had to spin on a CAS instruction until it completed successfully. As the execution time (and the amount of work that needs to be done) grows exponentially with the increase in size, the y-axis is set to a logarithmic scale.

The first experiment, using the thread pool guarded by a mutex is shown as "FFT Mutex" in Figure 15 on page 18. The *T3-4* is roughly two times slower than the *X4470*, but this was to be expected after the results we saw in Section 6, showing larger thread creation and synchronization overheads. What is clear is that all three systems scale up well in this experiment, roughly doubling their execution time for every increase in FFT size, as expected.

In the second experiment, we use the thread pool that is updated by atomic CAS instructions. For the *T3-4* system we use the ***atomic_cas_ptr()*** function that is defined in ***atomic.h***, and on the Linux systems we use the GCC builtin ***__sync_val_compare_and_swap()*** which is x86 specific. Before we made this experiment we thought it would perform better, as there would be less contention without a global lock on the pool. In earlier experiments on single and dual core systems, this implementation gave us a speedup of around 5%. But, as can easily be seen in Figure 15 on page 18, this was not the case at all. The figure shows both the execution time for different FFT sizes, as well as a count of the total number of times all threads had to spin on a CAS instruction until they have successfully made an update on the pool. As this is counted with a non-atomic increment of a *volatile* global variable, the number is not guaranteed to be 100% accurate, but it gives us sufficient insight in what happens.

When we look at the lockless results for the *T3-4* in Figure 15 on page 18, we see that the execution time and the number of CAS spins really explodes. Taking into account that the y-axis in the plot is already logarithmic, it seems to increase double exponentially. After 5 iterations, we decided to give up the experiment as the next iteration (if it would complete at all) would potentially take a whole day or longer, and the measured data already gave us enough insight in the behavior on the *T3-4*. We then repeated the experiment on the *X4470* and the *Magny Cours*, which showed roughly similar behavior, with the *Magny Cours* requiring more CAS spins. Also both x86 systems did not completely explode in CAS spins as the *T3-4* did, this is probably because the *T3-4* has many more hardware threads, and we suspect that at some point up to 512 threads were spinning on a CAS operation where only one of them would succeed each time. On the *Magny Cours* this could only be 48 threads, and on the Hyperthreaded cores of the *X4470* it could be even only

FFT Execution time

	T3-4			X4470			Magny Cours		
Size	Mutex	Lockless	CAS spins	Mutex	Lockless	CAS spins	Mutex	Lockless	CAS spins
1	2.30	8.31	16025.00	1.05	8.36	75746.00	1.89	25.23	1008344.00
2	4.27	31.42	31613.00	1.93	817.06	784639.00	3.87	936.88	19131369.00
3	8.20	108.25	24149.00	3.52	9077.68	6200028.00	7.33	6249.76	18777849.00
4	16.24	1939.78	153195.00	7.08	21997.67	9520188.00	12.48	21881.38	24341609.00
5	33.69	1070967.16	31309089.00	14.29	61172.46	16082589.00	25.66	81966.83	35129041.00
6	73.77	-give up-	-give up-	30.08	154516.11	20955098.00	49.65	178624.43	39667675.00
7	156.56			62.07	308675.11	21976732.00	98.38	367934.62	72779717.00
8	342.75			130.38	819729.10	40079177.00	208.48	853270.19	49921929.00
9	760.56			273.67	-give up-	-give up-	448.37	2133073.33	63399129.00
10	1619.71			608.83			924.51	12432475.87	121808471.00
11	3273.92			1233.43			1981.69	-give up-	-give up-
12	6822.80			2799.88			4274.26		
13	13937.49			5992.06			8574.09		
14	29353.08			12549.83			18274.67		
15	61331.68			26136.08			38867.67		
16	126745.60			63186.77			78966.23		

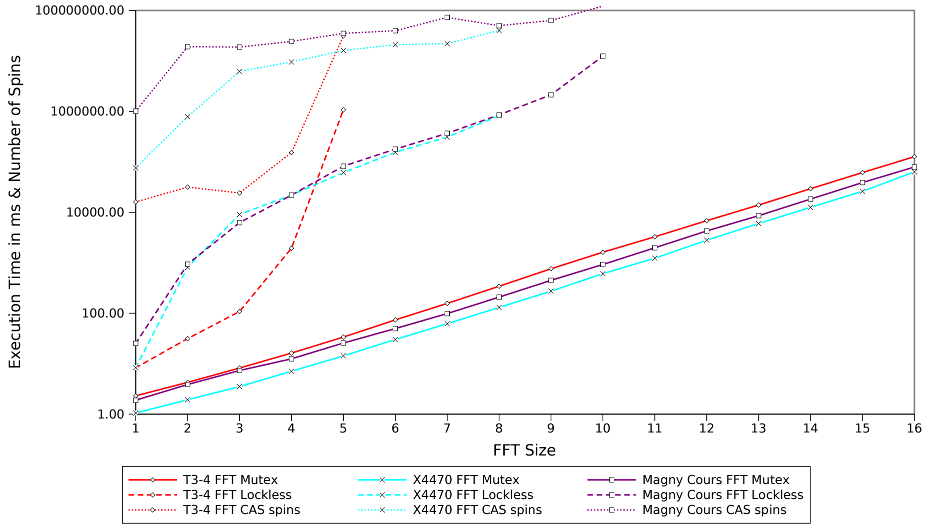

Figure 15. Table and plot showing results of FFT experiment

Matrix Multiply

T3-4

# Recursions	1024x1024	2048x2048	4096x4096	8192x8192
1	6920.19	87213.51	693818.43	5545460.11
2	1332.21	11273.06	89856.02	704318.33
3	827.80	3860.55	20779.43	127199.92
4	2602.63	5677.28	22903.12	126680.23
5	16999.49	22378.46	44356.10	166715.08

X4470

# Recursions	1024x1024	2048x2048	4096x4096	8192x8192
1	13488.27	107785.98	853244.23	6824218.89
2	3307.57	24202.94	186878.51	1488379.95
3	3453.76	24942.95	172086.54	1345201.05
4	7481.98	35142.30	192747.18	1391505.31
5	229046.81	194483.90	249475.48	1469648.87

Magny Cours

# Recursions	1024x1024	2048x2048	4096x4096	8192x8192
1	10598.72	85264.44	680671.55	5430166.97
2	2540.31	16591.91	129087.18	1019627.55
3	2859.07	15979.14	117799.68	913045.00
4	13538.94	27999.53	134528.84	953823.76
5	437244.33	494179.50	196412.36	970080.40

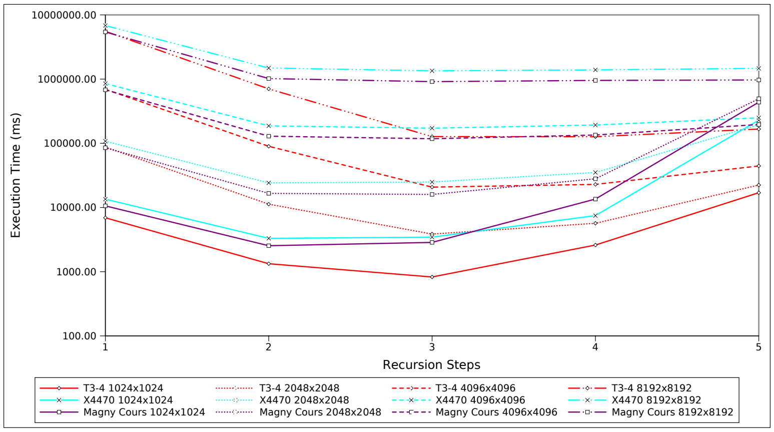

Figure 16. Table and plot showing results of recursive Matrix Multiplication experiment

24 threads, explaining the difference between the three systems. Another reason that makes the situation escalate is that when the run-time can't get a worker thread from the pool it will start a new pthread, which after completing it's work will also start to compete with CAS spins to enter itself into the pool again. This is probably the cause, together with the massive number of hardware threads, that causes the observed behavior on the *T3-4*.

Another interesting experience we had during the FFT experiment of size 5 on the *T3-4* was that the system appeared to completely hang for a while (around 2 minutes), making us initially think that we had crashed it. However, it still responded on the network to ping command's ICMP packets, and after a long delay our remote login shells started to respond again. Probably the system was completely swamped with coherency traffic from all the CAS operations, something we did not experience this heavily on the two other reference systems. Again, this difference is likely related to the great number of hardware threads offered on the *T3-4*.

7.2. Matrix Multiplication

In the second experiment based on the SVP-ptl implementation, we use a more suitable granularity of threads for this specific SVP implementation. We run an SVP implementation of a matrix multiplication using recursive block decomposition. Both $N \times N$ matrices are divided into four sub-matrices of $\frac{N}{2} \times \frac{N}{2}$, resulting in eight concurrent multiplications of these followed by four matrix additions. Of course the same can be applied recursively to each of the eight multiplications of sub matrices, yielding 8^k concurrent activities where k is the number of recursions. At the inner recursion when two matrices of $n \times n$ are multiplied, this results in $2 \cdot n^3$ multiplication and addition operations. As we run the experiments for $1 \ldots 5$ recursions and with N ranging from 1024 to 8192, the smallest threads are created in the 5th recursion on a 1024×1024 matrix, which still yields $2 \cdot (\frac{1024}{2^5})^3 = 65536$ floating point operations per thread. The largest threads occur with a single recursion on a 8192×8192 matrix, which then yields $2 \cdot (\frac{8192}{2^1})^3 = 1.37 \cdot 10^{11}$ floating point operations per thread. In this experiment we have the default mutex based thread pooling enabled, and we leave the mapping of the threads up to the operating system.

We ran our matrix multiplication code on all three systems and compare the execution time for different matrix sizes and number of decomposition recursions in Figure 16 on page 19. Surprisingly enough, the *T3-4* outperforms both the *X4470* and *Magny Cours* for the smallest matrix size, with only one recursion. This was not what we would expect after our earlier experiments on computational throughput, where these two systems clearly outperformed the *T3-4* with 8 computational threads (see Figure 4 on page 7). However, in contrast to that experiment, this has a more regular computation which can also be expressed as a multiply-accumulate for which the *T3-4* has a dedicated instruction which is not present on current x86 architectures. This might explain why the *T3-4* outperforms the two reference systems for all matrix sizes with a single recursion.

The *T3-4* scales up well to 3 recursions, which is to be expected as this corresponds with a maximum of $8^3 = 512$ concurrent threads. The optimum for the two reference systems lies clearly at 2 recursions, corresponding to a maximum of $8^2 = 64$ threads, which lies the closest to the number of hardware contexts (48) both systems have. Interestingly, for large matrix sizes the deeper recursions don't seem to give much penalty on the reference systems, but for the 1024×1024 and 2048×2048 matrices this is not the case, and at 5 recursions (yielding $8^5 = 32768$ threads at a maximum) they suffer a lot from overhead. The *T3-4* manages this relatively well; 5 recursions gives a 20-fold slowdown on a 1024×1024 matrix compared to the optimum at 3 recursions, against a factor of 69 on the *X4470*, or 174 on the *Magny Cours*. For the largest matrix multiplication, 8192×8192, we really see the *T3-4* take the advantage of its larger computational- and memory throughput, which we already demonstrated in Section 4 and Section 5.

7.3. Conclusions

The *T3-4* system seems to scale up to a lot of threads very well, and performs very good in our matrix multiplication experiments probably due to the hardware multiply-accumulate support. However, when a lot of threads are using atomic compare-and-swap, it becomes a nightmare, even worse than the x86 based reference systems. It would be interesting to investigate if the other atomic operations defined by ***atomic.h*** suffer from the same effect, and hence should be avoided. The advantage of using synchronizations of the pthread library such as mutexes, is that it will suspend a thread to the OS scheduler instead of having a thread spin in retries making the situation worse.

8. Overall Conclusion

In this report we have tried to explore the *T3-4* system from our own research's perspective, and we compared it to two other x86-based multi-core systems that we had available. We have investigated the scalability limits of both raw computing and memory throughput, where the *T3-4* showed very good scalable performance which amazed us. It really takes 512 threads to saturate the integer or floating point arithmetic units, and up to 256 threads to saturate the memory read bandwidth which outperforms the reference systems by a factor two. Especially that last fact helps the architecture to live up to the promises about CMT in the T3 Whitepaper [2]; as plenty of memory bandwidth is required to feed the many threads of execution. The experiments confirmed these nice dataflow like scheduling properties between threads in one core, with very fine-grained hardware multithreading and latency hiding.

Our experiments with mapping threads to specific cores showed that Solaris is well capable of distributing the work across all cores in a nearly optimal way by itself. We also revealed that creation of threads and synchronizations are still expensive in Solaris, and are outperformed by the x86-based reference systems that run Linux. However, the *T3-4* seemed to deal much better with extreme amounts of threads than the other two reference systems. Finally, we discovered that atomic operations can be very tricky on such a system with that many threads actually executing in hardware in parallel, possibly ending up in nightmare scenarios as we had the *T3-4* system hanging for 2 minutes. The x86 based systems showed similar problems, but due to the much lower number of hardware contexts, (more than a factor of 10), the problem was not as prevalent.

References

[1] J. Shin, K. Tam, D. Huang, B. Petrick, H. Pham, C. Hwang, H. Li, A. Smith, T. Johnson, F. Schumacher, D. Greenhill, A. Leon, and A. Strong, "A 40nm 16-core 128-thread CMT SPARC SoC processor," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pp. 98–99, February 2010.

[2] Sun/Oracle, "White paper: Oracle's SPARC T3-1, SPARC T3-2, SPARC T3-4 and SPARC T3-1B Server Architecture." `http://www.oracle.com/technetwork/articles/systems-hardware-architecture/sparc-t3-server-architecture-176017.pdf`, October 2010. (Retrieved January 2011).

[3] Sun/Oracle, "SPARC T3 processor Data Sheet." `http://www.oracle.com/us/products/servers-storage/servers/sparc-enterprise/t-series/sparc-t3-chip-ds-173097.pdf`, 2010. (Retrieved January 2011).

[4] Sun/Oracle, "SPARC T3-4 server Data Sheet." `http://www.oracle.com/us/products/servers-storage/servers/sparc-enterprise/t-series/sparc-t3-4-ds-173100.pdf`, 2011. (Retrieved June 2011).

[5] Sun/Oracle, "Sun Fire X4470 Server Data Sheet." `http://www.oracle.com/us/products/servers-storage/servers/x86/sun-fire-x4470-ds-079894.pdf`, 2010. (Retrieved January 2011).

[6] Intel, "Product Brief: Intel Xeon Processor 7500 Series." `http://www.intel.com/Assets/en_US/PDF/prodbrief/323499.pdf`, 2010. (Retrieved January 2011).

[7] Dell, "PowerEdge 11G R815 Spec Sheet." `http://www.dell.com/downloads/global/products/pedge/en/poweredge-r815-spec-sheet-en.pdf`, 2010. (Retrieved January 2011).

[8] AMD, "AMD Opteron 6000 Series Platform Quick Reference Guide." `http://sites.amd.com/us/Documents/48101A_Opteron%20_6000_QRG_RD2.pdf`, 2010. (Retrieved January 2011).

[9] C. Jesshope, M. Lankamp, and L. Zhang, "Evaluating CMPs and their memory architecture," in *Proc. Architecture of Computing Systems* (M. Berekovic, C. Muller-Schoer, C. Hochberger, and S. Wong, eds.), pp. 246–257, 2009.

[10] K. Bousias, N. Hasasneh, and C. Jesshope, "Instruction level parallelism through microthreading—a scalable approach to chip multiprocessors," *Comput. J.*, vol. 49, pp. 211–233, March 2006.

[11] C. Jesshope, "A model for the design and programming of multi-cores," *Advances in Parallel Computing*, vol. High Performance Computing and Grids in Action, no. 16, pp. 37–55, 2008.

[12] M. W. van Tol, C. R. Jesshope, M. Lankamp, and S. Polstra, "An implementation of the SANE virtual processor using POSIX threads," *Journal of Systems Architecture*, vol. 55, no. 3, pp. 162–169, 2009. Challenges in self-adaptive computing (Selected papers from the Aether-Morpheus 2007 workshop).

[13] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. van der Wijngaart, and T. Mattson, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *Solid-State Circuits Conference - Digest of Technical Papers, 2010. ISSCC 2010. IEEE International*, pp. 19–21, February 2010.

[14] GNU Project, "GCC, the GNU Compiler Collection." `http://gcc.gnu.org`, 2011. (Retrieved January 2011).

[15] Apple Inc., "Grand central dispatch, a better way to do multicore." `http://images.apple.com/macosx/technology/docs/GrandCentral_TB_brief_20090903.pdf`, August 2009. (Retrieved January 2011).

Acknowledgement

The author would like to thank Sun/Oracle for early access to a SPARC T3-4 system through the CMT beta programme which made these experiments possible, as well as the feedback given on the work and this report.

About the Author

drs. Michiel W. van Tol is currently a PhD student in his final year at the Computer Systems Architecture group at the University of Amsterdam, where he also received his MSc degree in 2006, specializing in Computer Architecture. He has a broad background and interest in computer systems, including nearly ten years of experience as a professional Linux System Administrator. His research interests include Many-core Architectures, Operating Systems, Parallel run-times and Multi-threading, Distributed Systems and Resource Management. Currently he is writing his PhD thesis on designing Operating Systems for SVP based Many-core architectures.