Threaded Dynamic Memory Management in Many-Core Processors

Edward C. Herrmann and Philip A. Wilsey
Experimental Computing Laboratory, Dept. of ECE, PO Box 210030, Cincinnati, OH 45221–0030

***Abstract*—Current trends in desktop processor design have been toward many-core solutions with increased parallelism. As the number of supported threads grows in these processors, it may prove difficult to exploit them on the commodity desktop. This paper presents a study that explores the spawning of the dynamic memory management activities into a separately executing thread that runs concurrently with the main program thread. Our approach works without requiring modifications to the original source program by redefining the dynamic link path to capture `malloc` and `free` calls in a threading dynamic memory management library. The routines of this library are setup so that the initial call to `malloc` triggers the creation of a thread for dynamic memory management; successive calls to `malloc` and `free` will trigger coordination with this thread for dynamic memory management activities. Our preliminary studies show that we can transparently redefine the dynamic memory management activities and we have successfully done so for numerous test programs including most of the SPEC CPU2006 benchmarks, Firefox, and other unix utilities. The results of our experiments show that it is possible to achieve 2-3% performance gains in the three most memory-intensive SPEC CPU2006 benchmarks without requiring recompilation of the benchmark source code. We were also able to achieve a 3-4% speedup when using our library with the gcc and llvm compilers.**

***Index Terms*—Many-core, threads, dynamic memory, SPEC benchmarks**

I. Introduction

Emerging multi-core and many-core processors provide opportunities for parallel execution of threads in emerging commodity desktop processors. While these solutions have been available for some time in the server market, only recently have such processors entered the low-cost consumer desktop market. Currently the desktop solutions contain mostly dual and quad core processors. However, the recently released Intel Core i7 has four hyper-threading cores (2 threads/core) which provides support for 8 concurrent threads and in 2007 Intel announced a processor prototype with 80 cores [1]. It is only a matter of time before commodity desktop computers contain many-core processors supporting tens to hundreds of threads per processor [2].

The migration of this much parallelism into the desktop may prove difficult to exploit fully. Certainly upgraded operating systems and major application programs such as web browsers and office tools can be reprogrammed with additional task level parallelism but even this may not provide sufficient parallelism for an (for example) 80 core processor. Furthermore, in the commodity market, many users will generally stick with non-upgraded applications when purchasing new desktop systems. Thus, techniques to transparently increase the parallelism of existing applications should be explored.

In this paper we examine the possibility of transparently threading the dynamic memory management activities of executing (system and application) programs. The general idea is to deploy a new library for dynamic memory management that forks a new thread for dynamic memory management. In this library, initial calls to `malloc` trigger the creation of a new thread and subsequent calls to `malloc` and `free` cause synchronization with and service from the spawned thread. Because most systems dynamically link to runtime libraries, the transparent integration of our dynamic memory management library is simply a matter of redefining the system link path to point at our library before the system memory management library. Of course the potential impact of this threading is only significant if there is a worthwhile amount of execution time costs in the dynamic memory management operations that can be hidden in overlapped execution with the original application thread. Thus, in this paper, we explore two issues. First, we examine the execution time costs of dynamic memory management in programs. In this study we will examine (primarily) the SPEC CPU2006 benchmarks [3]. Second, we have tested a threaded dynamic memory management library to gain some initial experiences and performance numbers. The library has been deployed on a Linux-based Intel i7 system and we have successfully used it to execute a variety of system and application programs, including the SPEC CPU2006 benchmarks, Firefox, and other unix utilities.

The remainder of this paper is organized as follows: Section II presents some of the previous work in optimizing dynamic memory management. Section III reports performance data showing the dynamic memory costs in the SPEC 2006 benchmarks. Section IV explains our implementation of a dynamic memory library to fork and manage dynamic memory threads. Section V presents the empirical results of using the threading dynamic memory management utilities. Section VI describes how this threaded technique could be applied to other libraries. Finally, Section VII presents a summary of our results and contains concluding remarks.

II. Related Work

One of the major issues with current dynamic memory allocation systems is that they do not scale well with multithreaded programs. Many of the current standard allocation systems were designed before multiple processor systems

Support for this work was provided in part by the National Science Foundation under grant CNS–0915337.

became commonplace [4]. As a result, memory concurrency issues were not factored into designing the systems. Since then, most allocators have been updated to support multi-threaded programs by using mutex locks around any dynamic memory function to ensure no race conditions occur. This is a simple solution, but it adds synchronization overhead to the critical paths of all allocations and deallocations [5]. The allocation functions can also become bottlenecks when multiple threads want to allocate or free memory at the same time. Although a program may be multithreaded, it essentially becomes sequential when threads must wait before entering atomic sections of code in the allocation functions [6].

Numerous memory allocators have been created to help alleviate this problem. Ptmalloc was an extension built on top of Doug Lea's dlmalloc that incorporates multiple regions in the heap so that threads can access different regions of the heap in parallel [7]. McIlroy *et al* created an allocator that used local core memory regions called scratch-pads to localize heap memory distribution [8]. Dice and Garthwaite [4] and Michael [7] proposed allocators that avoid using locking mechanisms. Hudson and associates created a lock-free allocator named McRT-Malloc [9]. Streamflow [5] is a scalable and locally conscious dynamic memory allocator. One of the most recent dynamic memory libraries for multithreaded applications is the `Hoard` allocator, created by Berger *et al* [10]. The `Hoard` allocator avoids global locking by creating thread-local heaps for each processor. These heaps can grow or shrink by taking blocks from a global heap area. This allocator also helps avoid false sharing, which occurs when values in separate caches share the same cache block [10].

All of these allocation systems do well in increasing the performance of multithreaded applications. However, not all applications are multithreaded. There are many legacy applications and programs that are not inherently parallel. Even native multithreaded applications have limitations as to how much parallelism can be exploited. There remains a need to find new ways to extract parallelism out of sequential programs. Threading common system code such as the dynamic memory allocation functions is one way to improve the performance of sequentially coded applications on multiprocessor systems.

III. Dynamic Memory Execution Costs

To show that performance improvements obtained by threading dynamic memory can be meaningful, it must be shown that the time spent in dynamic memory operations is significant. The cost of dynamic memory management is heavily program dependent. There are many factors that determine the dynamic memory behavior of a process: the frequency of allocations and de-allocations, the size of the requested blocks, the pattern in which the blocks are requested, and the order in which the requests occur. All of these attributes can also affect memory fragmentation, which can slow down the search times for future allocations. It is also difficult to obtain consistent measurements because program behavior can change for each run depending on the program inputs, external interrupts, and CPU scheduling of other processes.

fp_tests	**percent of total computation**				
	malloc	**realloc**	**calloc**	**free**	**total**
bwaves	0.00	0.00	0.00	0.00	0.00
gamess	0.00	0.00	0.00	0.00	0.00
milc	0.01	0.00	0.00	0.00	0.01
zeusmp	n/a	n/a	n/a	n/a	n/a
gromacs	0.00	0.00	0.00	0.00	0.00
cactusADM	0.00	0.00	0.00	0.00	0.00
leslie3d	0.00	0.00	0.00	0.00	0.00
namd	0.00	0.00	0.00	0.00	0.00
dealll	n/a	n/a	n/a	n/a	n/a
soplex	0.01	0.00	0.00	0.00	0.01
povray	0.03	0.00	0.00	0.02	0.05
calculix	0.02	0.00	0.00	0.01	0.03
GemsFDTD	0.01	0.00	0.00	0.00	0.01
tonto	7.52	0.00	0.00	5.96	13.48
lbm	0.00	0.00	0.00	0.00	0.00
wrf	0.17	0.00	0.00	0.09	0.26
sphinx3	0.00	0.00	0.23	0.06	0.29
int_tests	**percent of total computation**				
	malloc	**realloc**	**calloc**	**free**	**total**
perlbench	0.96	0.16	0.00	0.57	1.69
bzip2	0.00	0.00	0.00	0.00	0.00
gcc	0.19	0.00	9.68	0.28	10.15
mcf	0.00	0.00	0.00	0.00	0.00
gobmk	0.01	0.00	0.00	0.00	0.01
hmmer	0.02	0.02	0.01	0.02	0.07
sjeng	0.00	0.00	0.00	0.00	0.00
libquantum	0.00	0.00	0.00	0.00	0.00
h264ref	0.00	0.00	0.00	0.00	0.00
omnetpp	5.86	0.00	0.00	4.24	10.10
astar	0.21	0.00	0.00	0.12	0.33
xlanchbmk	2.26	0.00	0.00	1.24	3.50

$^{n/a}$Data for these programs were not gathered due to their extremely long callgrind runtimes (still running after 7 days)

Fig. 1. SPEC CPU2006 callgrind profiling results

In this work, we study primarily the SPEC CPU2006 [3] benchmark suite for our performance analysis. The SPEC benchmarks are processor intensive and consist of representative programs from a diverse range of applications. The SPEC benchmarks are widely used by the computer architecture community to study design tradeoffs. They also come complete with batch execution scripts to facilitate their use in testing and profiling system performance.

We use valgrind [11] to profile the runtime characteristics of our benchmark programs and to capture results of the computational costs that dynamic memory management adds to each benchmark program's total execution time. These results for the SPEC benchmark suite are summarized in Figure 1. This table lists the percent of overall assembly instructions executed inside each of the four main dynamic memory functions (`malloc`, `realloc`, `calloc`, and `free`). The sum

	percent of total computation					
Test	**description**	**malloc**	**realloc**	**calloc**	**free**	**total**
Firefox	load top 10 most visited websites	1.08	4.88	2.13	1.62	9.71
vlc	play 2 mins of 420p video	1.86	1.18	0.40	1.29	4.73
gcc	compiling my malloc library	26.37	0.10	0.24	0.10	26.81

Fig. 2. Linux applications callgrind profiling results

of these numbers represents the overall percent of execution time spent inside all dynamic memory functions, as shown in the last column. A small collection of standard, fairly CPU-intensive, Linux applications were also run through callgrind to gather additional test points. The profiling results for these applications are given in Figure 2.

The results show that the dynamic memory costs vary from application to application. The majority of the tests use dynamic memory sparingly and spend insignificant amounts of time in the standard memory functions. There are however three benchmarks that spend more than 10% of total processing time dealing with dynamic memory management, namely: `tonto`, `gcc`, and `omnetpp`. `tonto` is a quantum chemistry package, `omnetpp` is a network simulator, and `gcc` is the standard GNU C compiler [3]. These applications would benefit the most from a parallelized dynamic memory library. Throughout the remainder of the paper, we will be focusing on the performance of these three benchmarks, as well as some tests involving code compilation. The distributed dynamic memory strategy that we propose will focus on improving the performance of these applications by introducing concurrency into the memory management subsystem. The next section details our approach on how to implement such a system.

IV. Threading Dynamic Memory Management

When the basic `malloc` and `free` functions are invoked, the processing that is carried out can be split into two categories: *request processing* and *system processing*. Request processing is the minimal processing required to fulfill the current request. For a call to `malloc`, this includes finding a block of free memory of the adequate size and returning the address. For calls to `free`, the request-specific processing involves marking the block at the specified address as unused. System processing is the extra processing that takes place to help manage the system. This includes coalescing free blocks together, maintaining sorted linked lists of available blocks, and requesting new chunks of memory from the operating system. System processing maintains efficiency by keeping the level of internal and external memory fragmentation low, and also prepares the system for future requests by performing maintenance on internal data structures.

Both types of processing are essential to the successful and efficient operation of a memory management system. Current implementations of dynamic memory management systems distribute system processing across all the function calls. Requests for more memory from the O/S are made by `malloc` when no more free blocks are available. The `free` function combines newly freed blocks with adjacent free blocks as they arrive. Both functions maintain internal linked lists and other data structures that allow for quick indexing of freed blocks. In single processor systems this method of distributing the system processing alongside the request processing works very well. The main program has to be interrupted at some point to handle the system processing, so dividing the processing up evenly among the calls allows the system to carry out each request in a reasonable amount of time. In multiprocessor environments, the main program does not have to be interrupted to handle system processing. If properly implemented the system processing can be done in parallel with the main program, performing background cleanup and maintenance operations on the dynamic allocation system while the main program continues. The goal of our research is to create a plug-and-play dynamically linked library that will take advantage of this parallelism by having a separate thread take care of the system processing.

The first step was to override the default system free and malloc commands. Since we decided to use a Linux development environment, the `LD_PRELOAD` environment variable allowed us to easily override the default library search path. By creating a new dynamic library that redefined the dynamic memory functions and placing it earlier in the dynamic link chain, any dynamically linked program will use our replacement `malloc` and `free` functions. For our initial research, we did not want to completely recreate the entire dynamic memory system. Therefore, we built our memory management system on top of the original memory functions by allowing our library to act as a wrapper. The `dlsym` function recurses through the dynamically linked library structure to locate the next occurrence of a particular function. Using this command our library was able to create new chunks of memory by calling the system `malloc` command, and similarly was able to free memory using the system `free`.

Our first implementation (which we call *pass-through*) of the library created a separate memory manager thread upon the first invocation of `malloc` or `free`. We used the Native POSIX Thread Library (NPTL) functions to create and synchronize the manager thread with the program threads. Because the thread communication took so much time compared to the time spent in calling a single dynamic memory function, it was clear that another approach was needed. Our second implementation (called *distributed*) focused on minimizing the communication required between the threads. In our case, we minimized the communication by pre-allocating free blocks of predetermined sizes, so that they would be ready to be taken by

fp_tests	**% cost**	**sys malloc**	**lock free**
bwaves	0.00	14.90	14.20
gamess	0.00	14.50	14.50
milc	0.01	16.00	16.30
zeusmp	n/a	12.00	12.00
gromacs	0.00	6.43	6.43
cactusADM	0.00	8.71	8.24
leslie3d	0.00	8.93	8.50
namd	0.00	12.10	12.10
dealll	n/a	19.30	19.10
soplex	0.01	21.70	21.30
povray	0.05	15.30	15.20
calculix	0.03	4.47	4.47
GemsFDTD	0.01	10.30	9.72
tonto	13.48	11.10	11.50
lbm	0.00	32.60	30.90
wrf	0.26	n/a	n/a
sphinx3	0.29	26.30	26.30
int_tests	**% cost**	**sys malloc**	**lock free**
perlbench	1.96	19.50	19.80
bzip2	0.00	12.10	12.10
gcc	10.15	20.20	20.60
mcf	0.00	29.20	29.30
gobmk	0.01	16.60	16.40
hmmer	0.07	8.04	8.04
sjeng	0.00	16.50	16.50
libquantum	0.00	28.10	28.00
h264ref	0.00	21.10	21.00
omnetpp	10.10	16.60	16.90
astar	0.33	11.50	11.70
xlanchbmk	3.50	21.80	23.20

$^{n/a}$Data for the wrf benchmark was not gathered due to runtime errors in the benchmark.

Fig. 3. SPEC benchmark result summary

the next call to `malloc`. Pre-allocated memory blocks were sized using powers of two and stored in “bins”, which were linked lists of blocks of a particular size. On the free side, addresses of freed blocks of memory were stored in a circular array to be freed by the memory manager thread.

Results show that our distributed approach performs much better than the pass-through implementation, but that it remains slower than the original non-threaded library. This led us to a third implementation that uses a lock-free communication exchange which we describe more fully below.

In our third implementation (called *lock-free*), atomic operations are used to ensure data integrity between threads. To protect access to the free blocks, each bin has a counter that records the number of free blocks available. Prior to each allocation, this counter is atomically decremented. If the value returned from the atomic decrement function is greater than zero, then a free block is reserved for that allocation and is guaranteed to be available. An atomic compare and swap (CAS) instruction is used to remove the address from the front of the bin. When adding new blocks into the bins, the memory manager also calls an atomic decrement to reserve the last block in the bin. Once the last block is reserved, the new linked list of free blocks is appended to the list. Once the new blocks are added, the counter is atomically incremented to include the new quantity of blocks added to the bin. On the free side, a similar counter is used to assign entries in the address array to each incoming free request. Atomically incrementing the counter ensures that no two free requests will reserve the same array index. The manager thread travels through the free array freeing any entry that has a valid address.

In addition to removing locks, additional performance improvements were made to the lock-free algorithm. In particular, to prevent blocking during free requests, the free address array was replaced by a linked list. Each recently freed block stores a pointer to the next block in the list. Besides preventing blocking on the free requests, this also allows the manager thread to free the blocks without having to traverse the entire address array to check for addresses. The reuse of freed blocks was also implemented. Instead of releasing a block of memory back to the system and having to recreate a block of the same size later, freed blocks are placed directly back into their respective bins (when the bins are not full). This requires assigning extra bytes along with each block to store the size of the block. When the block is freed, these bits are used to identify which bin to place the block. Freed blocks are placed back into the front of the bins to provide better temporal locality for future requested blocks.

V. Experimental Results

To measure the performance of our threaded dynamic memory implementations, the SPEC benchmarks were executed using each version of the library. The SPEC benchmark suite evaluates system performance by measuring the execution time of a specific benchmark and comparing it to the time taken by a reference machine. Each test outputs the ratio of the current system execution time to the reference system execution time, resulting in the benchmark score. All three versions of the library were tested but only results for the lock free implementation are presented (Figure 3). The data shows the percent of dynamic memory in the original program and the SPEC performance numbers with and without our lock-free library. These performance results are from the standard SPEC scripts and the reported number is the geometric mean of scores of ten trials. In each of these tests, the variance was below .003. Lastly, the test system hardware was an 2.66GHz Intel Core i7-920 processor with 3GB of RAM.

Results from the three most memory-intensive benchmarks are also displayed in Figure 4 (including data from all three of our implementations). In these results, system malloc is the Linux `glibc` ptmalloc library, which is a modification of Doug Lea’s dlmalloc. Although the first (pass-through) implementation performs comparably to the standard library

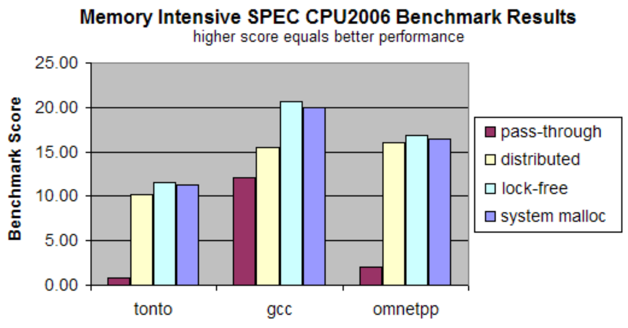

Fig. 4. Memory-intensive SPEC benchmark results

(system malloc) on the less memory intensive benchmarks, it produces the worst results on the memory intensive benchmarks due to the communication overhead involved. The second (distributed) solution shows much better results on these tests, but it is still not able to surpass the performance of the unthreaded library. The third (lock-free) method is able to provide performance gains of 2-3% in the benchmarks that have larger dynamic memory costs.

Analysis of callgrind results of the three memory intensive benchmarks shows that the number of instructions executed per call to `malloc` and `free` were reduced on average by 49% and 62%, respectively, with the lock-free library. Ideally then, we could have seen a potential maximum speedup of between 5-8% on these three benchmarks. Of course, cache effects, O/S threading and scheduling overheads, and other costs prevent reaching the ideal numbers.

Lastly, we show results with all of the SPEC tests, even those with minimal dynamic memory costs, to illustrate the possible negative costs of this approach. Clearly one would not use our approach for programs with minimal dynamic memory usage and instead only point the dynamic loader to our threaded library for applications containing larger amounts of dynamic memory usage.

To conclude our study with the SPEC benchmarks, we measured the performance of our lock-free library when compiling the benchmarks using the `gcc` and `llvm` compilers. To measure the effects of our threaded library we measured the time it takes to compile the entire benchmark suite using both the regular system malloc library and our lock-free dynamic memory library. For each compiler, five runs were performed. Figure 5 shows the results from the trial runs and the average speedup gained by using the threaded library. The results show that our library outperformed the standard library by 3-4%. This speedup is obtained by simply inserting the new library into the link path before calling the compile script. No changes to the source code or hardware were necessary. This speedup comes no cost to the system by utilizing extra processing power that would have otherwise gone unused.

gcc		
Original time(ns)	*Threaded time(ns)*	*Speedup(%)*
620664871896	606460323756	102.3%
628267056925	603266265277	104.1%
635297406466	605694212411	104.9%
630458667011	607401427149	103.8%
635380815759	606422945462	104.8%
Avg difference:	24164728800	Avg: **104.0%**

llvm		
Original time(ns)	*Threaded time(ns)*	*Speedup(%)*
723584918236	691882133455	104.6%
724780429654	696974112396	104.0%
714829357879	693343381500	103.1%
729322211743	702887493433	103.8%
723332596904	695465632151	104.0%
Avg difference:	27059352296	Avg: **103.9%**

Fig. 5. SPEC 2006 Compile Times using the Threaded Dynamic Memory Manager with the GCC and LLVM Compilers

VI. Further Applications

The multithreaded approach used in our library can also be applied to other library functions to provide performance improvement gains in other areas. Dynamic memory functions are a good candidate because they are used in a variety of programs and because memory blocks are easily pre-allocated. However, there are other computation-intensive library functions that might benefit from multithreading. A library function must exhibit certain properties in order for a threading implementation to be considered. In particular these properties must be met: it must be side-effect free, it must not rely on external program/machine state information, and it must be "use" idempotent. For example, the malloc function will return a pointer to a memory container of acceptable size; *which container* may vary, but any suitably sized container is acceptable. Any library function that can satisfy these requirements could be a candidate for optimization through threading. One example could be the trig functions sine and cosine. Provided applications call these functions using arguments that follow a predictable pattern (for example calculating the sine function in fixed degree increments) then it would be possible to precalculate future calls.

There are many factors that determine if threading would actually benefit an application. First the amount of speedup that can be gained will be limited by the amount of time a program spends inside the particular functions. As shown above in the experiments with the threaded dynamic memory library, applications that do not use dynamic memory extensively will not see speedup as a result of using the threaded library. Another factor is the amount of overhead involved in communication between the main program and the manager thread. The main program must provide data to the manager thread to allow accurate prediction of future calls. Similarly the manager thread must transfer the pre-calculated outputs to the main thread so that they will be available when the function calls arrive. The amount of time saved in function calculation must be greater than the communication overhead in order for speedup to occur. Misprediction penalties will also affect performance. If the input to the next function call is mispredicted, the function output must be calculated in-line, essentially negating any gains that the predicted value would have provided. Finally the timing of the calls in the program can also affect how effective library threading will be. There needs to be time separating successive function calls to allow the manager thread to predict the next output. If there are many function calls in a row, the predicted values may not be ready in time. This would cause the main program to wait for the output to be calculated, reducing the amount of parallelism that can be extracted from the function.

VII. Conclusion

The results of our experiments show that it is possible to obtain speedup from traditionally single-threaded applications by handling the dynamic memory management in a separate thread. Our custom library was able to achieve 2-3% performance gains in the memory-intensive SPEC CPU2006 benchmarks without requiring recompilation of the benchmark source code. We were also able to achieve a 3-4% speed up of the benchmark compile time (using either gcc or llvm). Traditionally dynamic memory libraries have had to deal with the tradeoff between the complexity of the algorithm and the speed at which it operates. Multithreading the dynamic memory system opens up more processing power without negative performance effects on the main application thread. This allows more complex but efficient algorithms to be implemented, providing extra time savings not only from faster dynamic memory function calls but also from better cache performance of allocated blocks. The multithreading of dynamic memory allows traditionally single-threaded applications to transparently take advantage of extra processors provided by multi- and many-core architectures.

Emerging many-core processors provide unique opportunities and significant challenges for the parallel processing community. In particular, we need to rethink our conventional thoughts of parallelism where researchers/software designers pursue solutions whose success is measured by scalability with the number of processors/threads. We must begin to think of these processors/threads as free/cheap resources that should be exploited whenever possible to gain any (and every) speedup opportunity. In this paper, we examine the small (but measurable) overhead of dynamic memory and develop techniques to transparently parallelize some aspects of the dynamic memory functions. In our case, we were able to reduce the average number of machine instructions for each `malloc` and `free` call by 49% and 62% respectively. Unfortunately this does not ultimately provide a corresponding reduction in the total dynamic memory costs. In our experiments, we achieve only 30-40% of the maximum possible reduction of dynamic memory costs. Factors such as cache effects, O/S (threading) overheads, and synchronization costs can (and do) impede speedup.

With multi-core processing already here and many-core processing on the horizon, is it important for programmers and architects to shift their perspective on computing in order to take advantage of the benefits many-core processing will provide. As processors shift from serial to parallel execution, it is crucial that software adapts to support these changes. We must find ways to introduce parallelism into programs. Methods such as predicting arguments for and early invocation of future function calls that would not be feasible to implement in a single core system can now thrive on emerging multi-core architectures. Even recent advances in parallel processor architectures can be seen through our experimentation. Many of the tests detailed in this paper were run on an Intel core2duo processor as well as the i7. None of the tests provided speedup on the core2duo, whereas the improved multi-core architecture in the i7 (specifically the much lighter-weight synchronization primitives) allowed us to report modest speedup success with the threaded dynamic memory library. Future many-core architectures must provide even better support for fine-grained multithreading similar to the type found in our threaded dynamic library implementation. Operating system impacts must also be considered. We are beginning to evaluate our library with Solaris, and, we are also considering work with more exotic O/S's such as Barrelfish [12].

References

[1] Intel Press Release, Intel Corporation, "Intel research advanced 'era of tera'," Intel Press Release, Intel Corporation, Tech. Rep., Feb. 2007. [Online]. Available: http://www.intel.com/pressroom/archive/releases/20070204comp.htm

[2] Intel Corporation, "From a few cores to many: A tera-scale computing research overview," Intel Corporation, Tech. Rep., 2006. [Online]. Available: http://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf

[3] S. P. E. Corporation, "Spec cpu 2006," Jan. 2009. [Online]. Available: http://www.spec.org/cpu2006/

[4] D. Dice and A. Garthwaite, "Mostly lock-free malloc," in *Proceedings of the 3rd International Symposium on Memory Management*, 2002.

[5] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos, "Scalable locality-conscious multithreaded memory allocation," in *Proceedings of the 5th international symposium on Memory Management*, E. Petrank and J. E. B. Moss, Eds. ACM, 2006.

[6] D. R. Butenhof, *Programming with POSIX Threads*, ser. Professional Computing Series. Addison-Wesley, 1997.

[7] M. M. Michael, "Scalable lock-free dynamic memory allocation," *Conference on Programming Language Design and Implementation*, vol. 39, no. 6, 2004.

[8] R. McIlroy, P. Dickman, and J. Sventek, "Efficient dynamic heap allocation of scratch-pad memory," in *The International Symposium on Memory Management*, R. Jones and S. M. Blackburn, Eds. ACM, 2008.

[9] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. Hertzberg, "Mcrt-malloc: a scalable transactional memory allocator," in *Proceedings of the 5th International Symposium on Memory Management*, E. Petrank and J. E. B. Moss, Eds. ACM, 2006.

[10] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," in *The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, 2000.

[11] M. Behm, "Using valgrind to detect and prevent application memory problems," *Redhat Magazine*, vol. 15, 2006.

[12] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schpbach, and A. Singhania, "The multikernel: A new os architecture for scalable multicore systems," in *Proceedings of the 22nd ACM Symposium on OS Principles*, Oct. 2009.