Improving Scalability of OpenMP Applications on MultiCore Systems Using Large Page Support *

Ranjit Noronha and D.K. Panda
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{*noronha,panda*}*@cse.ohio-state.edu*

Abstract

Modern multicore architectures have become popular because of the limitations of deep pipelines and heating and power concerns. Some of these multicore architectures such as the Intel Xeon have the ability to run several threads on a single core. The OpenMP standard for compiler directive based shared memory programming allows the developer an easy path to writing multithreaded programs and is a natural fit for multicore architectures. The OpenMP standard uses loop parallelism as a basis for work division among multiple threads. These loops usually use arrays in their computation with different data distributions and access patterns. The performance of accesses to these arrays may be dependent on the underlying page size depending on the frequency and strides of these accesses. In this paper, we discuss the issues and potential benefits from using large pages for OpenMP applications. We design an OpenMP implementation capable of using large pages and evaluate the impact of using large page support available in most modern processors on the performance and scalability of parallel OpenMP applications. Results show an improvement in performance of up to 25% for some applications. It also helps improve the scalability of these applications.

Keywords: *System Area Networks, Clusters, OpenMP, Multithreading*

1 Introduction

With deeper processor pipelines showing limited gains and with heating and power concerns, modern micro-processors are now moving to multicore architectures to extract more performance from available chip area. As a result multithreaded applications may potentially exploit maximum benefit from a multicore architecture. OpenMP standard [22] for shared memory parallel programs was specifically designed to allow programmers to easily write multithreaded programs. With the advent of modern multicore architectures, the OpenMP programming paradigm may potentially become an important model for application writers.

*This research is supported in part by Department Energy's grants #DE-FC02-06ER25755 and #DE-FC02-06ER25749, National Science Foundation grants #CNS-0403342 and #CNS-0509452; grants from Intel, Mellanox, and Sun Microsystems; and equipment donations from Intel, Mellanox, and SUN Microsystems.

Multicore architectures can largely be classified into different categories depending on how they are interconnected, as well as the sharing of the caches and the TLB's, etc. The Opteron based multicore processor for example implements two separate cores connected by separate hypertransport links and separate L2 caches. On the other hand the Intel Xeon implements separate cores on the same chip, which in turn share a common L2 cache. In addition, each core is capable of running up to two threads. As a result, the hardware resources of each core are shared between the two threads. This may create contention for resources such as caches and the translation lookaside buffer (TLB).

The OpenMP programming paradigm implements loop level parallelism, which is one one of the most basic available units of parallelism for parallel OpenMP programs. Loop-level parallelism allows an OpenMP implementation to easily split the work across multiple threads. Scalability of loop-level parallelism may depend on the division of work among different threads. In addition, for loops that do strided computation on an array of data, the size of the stride as well as the locality of the data may further limit the scalability of the threads in an OpenMP program. Larger stride patterns not only increase L2 cache misses, but also increase the data TLB misses.

The Translation Lookaside Buffer in modern processors is implemented to speed the translation of virtual to physical addresses. The translation is usually implemented through the multiple levels of page directories and tables stored in physical memory. Since accessing physical memory can take several hundred cycles depending on the architecture, the TLB can provide substantial gains in application performance. However, the TLB is a limited resource, and may not provide adequate benefit in the face of poor application locality. While traditionally, most pages have been 4K in size, modern processor also provide support for large pages, up to 2M or bigger. This could help dramatically reduce TLB misses and have an impact on the performance of a wide variety of applications.

Thus it is natural to ask whether OpenMP applications which exploit loop-level parallelism which perform strided access to several arrays, where the stride size is greater than a single 4K page can potentially benefit from large pages. This benefit would potentially come about from the reduction in TLB misses and a decrease in processor pipeline stalls.

In addition, to strided access, the TLB is shared across multiple threads running on the same core for architectures which implement SMT (Simultaneous Multi-threading). An example of such an architecture is the Intel Xeon processor which implements hyperthreading. Because of the shared nature of the TLB, depending on the access patterns and locality of the application, the number of entries in the TLB may be potentially halved. Large pages may potentially provide additional benefit in this case and may also have an impact on the scalability of the multithreaded application.

In this paper, we discuss the issues and potential benefits from using large page support for parallel OpenMP applications. We design an OpenMP implementation which can take advantage of large pages. We evaluate this implementation across a range of applications and multicore architectures. These evaluations show that there is an improvement in parallel performance of 25% for CG. In addition, the applications scale better. We also used instrumentation tools to better understand how large page support impacts the TLB cache misses. These results show a substantial reduction in TLB misses.

The rest of this paper is organized as follows. In section 2 we look at background material. Section 3 discusses potential issues and design challenges. In section 4, we look at the impact of large pages on a variety of different applications. Section 5 discusses related work. Section 6 discusses conclusions and future work.

2 Background

In this section we discuss multicore architectures and the OpenMP programming model.

2.1 Multicore Architecture

Multicore architectures have been evolving as a means to increase processing power, while reducing the impact of heating and power consumption, in addition to addressing some of the limitations of a super-scalar architecture. Multicore processors implement chip level multithreading (CMT). CMT allows multiple different threads of execution on the same processor. CMT can potentially improve performance of multithreaded workloads. Multicore architectures in general consist of several processor core on the same chip die. These cores usually share buses and caches. Several different variations of multicore architecture have been proposed and implemented. We briefly describe some of these architectures below.

Chip Level MultiProcessing (CMP): This technique implements separate processing cores for each thread on the die. Each core has an individual copy of the processor hardware. The processing cores are usually connected through a hardware bus for communication and may also share a cache. Examples of implementations of CMP's are the AMD dual-core Opteron processors. The cores on the Opteron are connected by hyper-transport links and have separate 1MB L2 caches which are kept coherent through snooping by the individual processors. Each processor in turn has a two level data Translation Lookaside Buffer (TLB) cache. The L1 DTLB has 128 entries, while the L2 DTLB has 1024 entries.

Symmetric Multithreading (SMT): Simultaneous multithreading (SMT) enables a single core to run one or more threads simultaneously. This is usually achieved through the use of multiple thread contexts on the same processor. The threads share different execution units and the processor is responsible for hazard detection and management between the different threads. Different implementations of SMT are possible. One potential implementation is to flush the pipeline on a thread stall and switch in the other thread. Hyperthreading implemented on the Intel Xeons is an example of this implementation. The other possibility is to implement different thread contexts and allow different stages of the pipeline to run different thread contexts. This potentially maximizes throughput, especially in the face of load stalls. The Sun Niagra is an example of this type of implementation.

CMT+SMT: The combination of CMT and SMT allows individual processor cores to run multiple threads. The Intel Xeon and Sun Niagra processors [23] are examples of this type of processor.

2.2 The OpenMP Programming Model

The OpenMP specification [8] for multi-processing is an API which may be used to direct multi-threaded, shared memory programs on SMP's. This is through the use of explicit compiler directives. It is based on the *Fork - Join* model of parallel execution as shown in Figure 1. OpenMP programs usually begin execution as a single process which contains a *master thread*. The master thread may execute sequentially till a parallel region is encountered, at which point multiple worker threads are spawned to process the parallel region. On completion of the parallel region, the threads exit and the master thread completes execution. To provide more fine-grained parallelism, directives for making variables, shared

or private may also be specified. In addition, features such as *Nested Parallelism* and *Dynamic Threads* may also be supported. Most implementations of OpenMP are compiler based and include those for the Intel compiler [1], Polaris [9], GCC [7] and Omni [3].

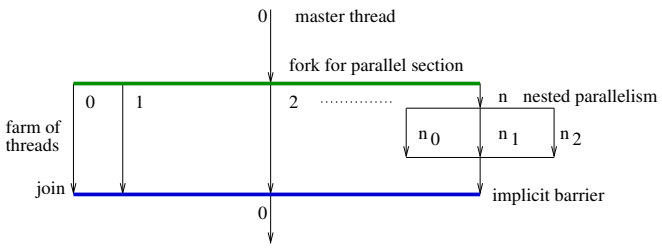


Fig. 1. OpenMP fork-join model

3 Challenges, Design and Implementation Issues of Multithreaded applications on shared memory systems with large page support

In this section, we look at the potential issues in using large page support for parallel OpenMP application. Section 3.1 discusses the potential impact of large page support on loop parallelism in OpenMP applications. Section 3.2 looks at the potential impact of the page tables and TLB sizes on parallel application performance. Finally, section 3.3 looks at some design and implementation issues.

3.1 Loop Level Parallelism in OpenMP

OpenMP offers a number of different directives for loop-level parallelism. It is possible to extract substantial gains by dividing up the work available in a loop among the different threads or processes in an application. Algorithm 3.1 shows a simple example of an OpenMP loop which sums the values of an array. If the array is very large, it might occupy several 4K pages in physical memory. As a result, all the threads in the parallelization phase may experience several TLB misses to access the array data. Using a larger page size has the potential to substantially reduce the number of TLB misses for all threads, and help improve performance. In addition, more complicated array accesses may occur in programs, such as strided accesses. These typically occur in codes for some algorithms of the Fast Fourier Transform [11]. Depending on the stride size, TLB misses might be a substantial performance penalty. We now discuss the issues and challenges involved in using large pages with parallel applications.

Algorithm 3.1: SUM(S)

$\#pragma\ omp\ parallel\ for\ private(i)$
$for(i = 0; i < S; i++)$
$sum+ = array[i];$

3.2 Page Tables and the Instruction and Data TLB's

Modern architectures support up to 64-bit memory address spaces. This allows for a total virtual memory area much larger than the physical memory of most modern computers. Since most applications in reality are likely to use only a fraction of their actual physical memory, most modern processors support the translation of virtual addresses to physical addresses. These translations are supported through segmentation and paging. The virtual address space of the operating system and applications is mapped through a series of tables to the actual physical address. These tables are usually managed by the operating system. An example of a page table architecture is shown in Figure 2. Figure 2 shows a three level page table architecture. Each process on a modern Linux system contains a *Page Global Directory (PGD)*. The PGD is the first level page table. It contains pointers to the middle level page table called the *Page Middle Directory (PMD)*. The entries in each PMD in turn point to individual page frame which contain *page table entries (PTE)*. The Linux kernel does not implement PMD for the x86_64 architecture, and only supports PGD's which point to page frames containing PTE's. On Linux, the virtual address is divided into three components. The leftmost bits are used to index into the PGD, the middle set of bits are used to index into the page frames containing the PTE's and the rightmost set of bits being used as an offset to the location in the physical page. The process of translating a virtual address to a physical address by traversing the PGD and page frames containing PTE's is called the page walk. The PGD and page frames containing the PTE's are stored in main memory. As a result, translating a virtual address to an actual physical address is an expensive operation, requiring two memory accesses. To speed this process, most modern processors implement a Translation Lookaside Buffer Cache (TLB). The TLB is usually split into an instruction TLB (ITLB) and a Data TLB (DTLB). Depending on the architecture, the TLB may be a two level architecture, as in the case of the Opteron processor (L1DTLB and L2DTLB).

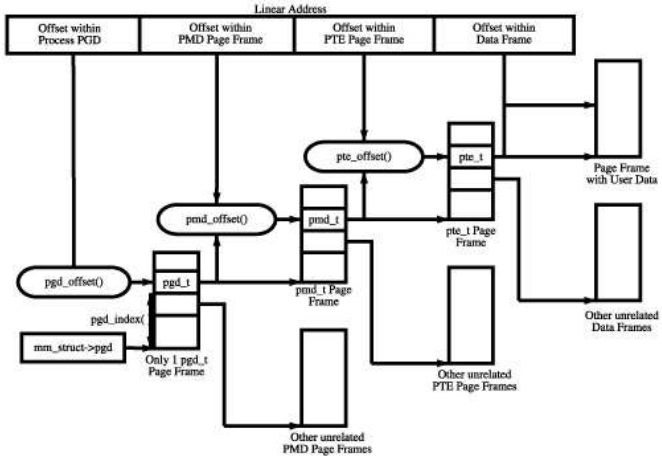


Fig. 2. Page table architecture, courtesy Gorman [19]

TLB Sizes and Memory Coverage: Both the Intel and Opteron processors have separate cache for data and instruction page and directory translations. Table 1 shows some of the TLB Sizes and memory coverage for the Intel Xeon and Opteron processors. These sizes were measured through the CPUID instruction [14, 10]. Most modern processors also support large 2MB pages in addition to the traditional

4KB pages. The ITLB and DTLB usually also have specific entries to support large pages. Since the entries for large pages may be different, the TLB's usually support a smaller number of entries for large pages. This is illustrated in Table 1. The Intel Xeon processor has 128 entries for 4K pages and 32 entries for 2M pages. Similarly, the Opteron processor has 32 entries for 4K pages in L1DTLB and 8 entries for 2M pages in D1TLB. The D2TLB in the Opteron does not have any entries for large pages. While the relative sizes of the pages and coverage is different, the large difference in TLB size can have an important impact on application performance, particularly for applications with poor locality.

Table 1. Processor TLB Sizes and Coverage

	Xeon	Opteron
ITLB (4K) Size	128	32
L1DTLB (4K) Size	128	32
L1DTLB (2M) Size	32	8
L2DTLB (4K) Size	–	512
L2DTLB (2M) Size	–	–
L2DTLB (4K) Coverage	512K	128K
L2DTLB (2M) Coverage	64M	16M

Application Locality and Large Pages: Though, the memory footprint is much higher in the case of 2MB pages, the smaller size of the DTLB for large pages might be a limitation in the case where the application makes multiple non-contiguous stride accesses with a stride access of larger than 2MB. With applications written in this way with lower spatial locality, the higher capacity of the DTLB for 4K pages might yield improved performance. This issue also occurs on the Opteron processor which has an L2DTLB size of 1024 for 4K pages and none for 2M pages. Applications with stride access larger than 2MB on the Opterons might in fact benefit more because of the larger L2DTLB.

SMT and the shared DTLB: In an SMT based processor system, the hardware resources are shared. This includes the DTLB. Parallel shared memory OpenMP programs may potentially exploit the multiple potential processor contexts available for improved performance. This could potentially result in two or more threads being scheduled on the same processor core. As the threads share the DTLB, depending on the access patterns of the application, the effective number of TLB entries could potentially be halved. For applications with good data locality and accessing more than 2MB sequentially, the impact of L2DTLB misses might be more severe. Using large 2MB pages may help reduce the frequency and impact of these misses.

SMT DTLB Context Switching Time: Large pages may potentially improve the performance of a multithreaded application on an SMT system. However, the limited number of DTLB entries for large pages in the processor cache may potentially become a bottleneck and reduce performance. In addition, memory load stalls typically evict the thread context. Depending on the design of the processor architecture, this context switch might dominate the application execution time. As a result, the potential

performance improvement from reduced DTLB misses might not translate into improved performance at the application level.

3.3 Design and Implementation Issues of OpenMP implementation for large pages

To measure the impact of using large pages on the performance of parallel applications, we use a modified version of the Omni/SCASH Cluster OpenMP implementation [3, 4, 20]. The Omni compiler transforms a C or FORTRAN program into multithreaded code. To enable it to work with the underlying SCASH DSM system, all global variable declarations are made into global pointers which are mapped to shared regions in the process memory space. For processes within a node (intra-node), the shared region is maintained via a memory mapped file. For processes on physically separate nodes, the underlying SCASH DSM system is responsible for maintaining data coherency. This is largely through the use of page memory protections on the shared regions which trigger a page handler, which is responsible for fetching the page from a remote manager. Our interest in using this system lies mainly in the global transformation of global data to a common memory mapped region. Omni/SCASH translates all global static variables and dynamic memory allocations into calls to its internal memory allocator. This memory allocator in turn allocates memory from the memory mapped file for processes on the same node. This allows all processes to in turn share the same memory image. We do not use any of the cluster OpenMP features of the SCASH DSM layer and instead only use it on a single node with multiple processes running. We briefly discuss some of the design challenges and tradeoffs in designing an OpenMP implementation for large pages.

Large Page Allocation: There are several studies for allocating large pages for applications on-demand and based on the allocation size [21, 26, 28, 25]. These strategies maximize the benefit of large page support when there are several different applications running on the system, and competing for memory. When running an OpenMP parallel application on a node, it is likely to be the only application running at the time. As a result preallocation of large pages is likely to reduce the complexity of the allocation algorithm and also the latency of the allocation. This will probably yield higher performance for the application. In addition, the Omni/SCASH cluster OpenMP implementation allocates both global shared and dynamic memory at process startup. This matches well with the preallocation of large pages. In our implementation, we preallocate a set of large pages which may be used by the processes through the hugetlbs [17]. We have modified Omni/SCASH to use the map file from a hugetlbfs file system [17]. All memory allocated in the hugetlbfs will use 2M pages.

Intra-node Communication: Omni/SCASH requires communication for the implementation of certain OpenMP primitives such as barriers, reductions, etc. The original implementation of Omni/SCASH use the SCore communication library [6]. SCore typically uses Myrinet [2] as the underlying communication substrate. We only use the intra-node SMP features of the Cluster OpenMP implementation and do not need a network for inter-node communication. To avoid having to use a Myrinet network interface, we implement a simple shared memory message passing interface through a file memory mapped into each processes space. The memory mapped file uses traditional small pages (4K) and not large pages. This implementation only uses a single memory copy (from the source process to the shared memory buffer). On the receiving process, the buffer may be directly accessed without the need for an additional copy. Through a set of flag, the processes may signal the availability of a message for the remote process as well as allowing a buffer to be freed up. Multiple outstanding messages may be in flight

between a set of processes (upto 32 in the current implementation). Since the intra-node communication are all small messages (less than 1K), this implementation is feasible.

Memory Protection: The Omni/SCASH cluster OpenMP implementation memory protects pages as a mechanism to trapping accesses to pages. This allows for coherency mechanisms of the eager release consistency (ERC) protocol to to take effect. We only use the cluster OpenMP implementation in intra-node mode. In this mode, the memory is shared between the different processes on the node. The underlying hardware is responsible for maintaining memory coherency. As a result, the memory protection mechanism is not needed. We disable this in our version of the Omni/SCASH OpenMP implementation.

In the next section, we discuss the performance evaluation of the large page support with OpenMP applications.

4 Performance Evaluation

In this section, we discuss the performance evaluation of parallel OpenMP applications with large page support. Section 4.1 discusses the hardware setup used to evaluate the applications. Section 4.2 discusses some of the characteristics of the applications we are using. Section 4.3 discusses the impact of instruction TLB misses on the performance of the application. Section 4.4 discusses the impact of large pages on application data TLB misses.

4.1 Experimental Setup

To evaluate our design, we use two hardware platforms. The first hardware platform consists of an dual, dual-core Opteron 270 processors (4-cores), with 4GB main memory and running SuSE Enterprise Linux. The other platform is a dual, dual-core Intel Xeon processor (4-cores) with hyperthreading enabled (enabling each core to run up to 2 threads for a total of 8 threads). The Intel Xeon system has 12GB of main memory and runs Redhat AS4. Both systems have a 2.6.19 kernel.org kernel which is multicore- and hyperthreading-aware.

4.2 Application Characteristics

In this section, we discuss some of the characteristics of the OpenMP version of the NAS Parallel Benchmarks [13] used in our evaluation.

BT: This is a computational fluid dynamic code that uses an implicit algorithm to solve 3-dimensional compressible Navier-Stokes equations. The finite differences solutions to the problem is based on an Alternating Direction Implicit (ADI) approximate factorization that decouples the x, y and z dimensions. The resulting systems are Block-Tridiagonal of 5x5 blocks and are sequentially along each dimension. BT consists of nine different arrays whose size depends on problem class chosen (A, B, C). Since the parallel portions of the loop process 5x5 blocks sequentially along each axes of type double (8 bytes), we can expect good spatial and temporal locality. In addition, a single block would potentially fit within a single page (200 bytes). However, approximately 20 blocks might fit in a single 4K page, while approximately 10485 blocks would potentially fit in a 2M page. As a result the number of TLB entries may be reduced. In addition, from Table 2, we can see that the total data footprint for class B is 371M. This comes out to only 186 2M pages, but 94,976 4K pages.

CG: This uses a conjugate gradient method to compute an approximation to the smallest eigenvalue of a large sparse, unstructured matrix. This kernel tests unstructured grid computations and communications by using a matrix with randomly generated locations of entries. Since the locations of the entries are randomly generated, the stride side may potentially be larger than 4K and 2M pages may potentially provide benefit to CG.

FT: This contains the computational kernel of a 3-D fast Fourier transform (FFT)-based spectral method. FT performs three one-dimensional (1-D) FFT's, one for each dimension. The FFT algorithm is based on the Cooley-Tukey algorithm, which divides the DFT of any composite size N=N1XN2 into many smaller DFT's of size N1 and N2 along with O(N) multiplications by complex roots of unity traditionally called twiddle factors. As a result the entire array along one dimension might fit in a single 4K page or multiple 4K pages, for which 2M pages might provide benefit. In addition, FT class B has a data footprint of 2.4 GB. As a result the number of data TLB entries just from the sheer size of the memory footprint is likely to be reduced by using 2M pages instead of 4K pages.

SP: This is a simulated CFD application that has a similar structure to BT. The finite differences solutions to the problem is based on a Beam-Warming approximate factorization that decouples the x, y and z dimensions. The resulting system has scalar pentadiagonal bands of linear equations that are solved sequentially along each dimension.

MG: This MG uses a V-cycle multigrid method to compute the solution of the 3-D scalar Poisson equation. The algorithm works continuously on a set of grids that are made between coarse and fine. It tests both short and long distance data movement. When the data movements tested are larger than 4K, 2M pages are likely to provide benefit. In addition, the large memory footprint of 884M (442 2M pages and 221K 4K pages) may potentially show a benefit from reduction in DTLB misses.

Table 2. Application Memory Footprint

	Instruction	Data
BT (B)	1.6M	371M
CG (B)	1.4M	725M
FT (B)	1.4M	2.4G
SP (B)	1.6M	387M
MG (B)	1.4M	884M

4.3 Impact of large pages on Instruction Misses of Parallel Application

Table 2 shows the sizes of the binary of the different NAS applications. As may be seen from the table, the binary size is slightly less than 2M. As a result, the binary may potentially fit in a single large page of 2M. This may potentially eliminate ITLB misses. By comparison the larger size of the ITLB in the Intel and Opteron processors using 4K pages may potentially cover close to 1/4 th of this memory area. Since most of the time in OpenMP applications are potentially spent in large parallel loops, we would expect

that the instruction temporal and spatial locality to be fairly high and the cost of instruction misses to be amortized across many accesses. Figure 3 shows the aggregate rate of instruction TLB misses for the applications BT, CG, FT, SP and MG running 4 threads on a dual dual-core Opteron platform, measured using the OProfile [5] tool. MG shows the highest rate of 0.45 instruction misses/second. With modern processors running at 2.0 GHz, assuming an ITLB miss of 200 cycles, this corresponds to a miss penalty of approximately 90 cycles/second. Thus, the ITLB miss rate is not likely to be a significant source of overhead, and may potentially not benefit from large pages.

4.4 Impact of large pages on Application Data Misses

In this section, we discuss the impact of large pages on application data in the parallel OpenMP applications. We first discuss the impact on system scalability, followed by the impact on data TLB misses.

System Scalability: Figure 4 shows the impact of small 4K pages and large 2M pages on the applications BT, CG, FT, SP and MG. We evaluate these applications on a dual-processor dual-core Opteron 270 system and on a dual-core dual processor Intel Xeon system with hyperthreading (SMT) enabled, allowing us to evaluate the system up to 8 SMT's. As can be seen from the Figure 4, the Intel and Opteron systems perform similarly on all five applications up to 4 threads. At 8 threads, the Xeon platform does not scale well. A similar observation was made by Chapman, et.al [16]. We attribute this to the implementation of SMT on the Intel Xeons which flush the entire pipeline on a thread context switch. This has considerable impact on the performance of the applications. We can make the following observations from Figure 4. Large page support has an impact on the performance of the applications CG, SP and MG. For CG, on the Opteron 270 based system, at 4 threads, there is an improvement of approximately 25%. On Opteron 270, SP shows a performance improvement of 20% at 4 threads with 2M pages over 4K pages. On the Intel Xeon's, SP shows a performance improvement of 13% at eight threads with 2M pages. In addition, while the SMT implementation on the Xeon's prevents SP from scaling from 4 to 8 threads, 2M pages help improve scalability from 2 to 4 threads. For MG, on Opteron 270, there is a performance improvement of approximately 17% at 4 threads with 2M pages. Large pages enable CG, MG and SP to scale better on both the Opteron and Xeon platforms. For applications BT and FT there is no significant improvement in performance when using 2M pages instead of 4K pages. We will now examine the impact of Data TLB misses on the performance of the applications.

Data TLB Misses: The OProfile [5] tool allows use to measure a number of different processor statistics. Using OProfile, we measured the number of DTLB misses on the Opteron platform for the different applications. Figure 5 shows the DTLB misses with 4K and 2M pages at four threads. The 4K and 2M misses were normalized with respect to the 4K misses for each of the applications. From Figure 5 we can see that for applications CG, SP and MG the number of DTLB misses is reduced by approximately a factor of 10 or more when using 2M pages over 4K pages. CG, SP and MG in turn show the most benefit when using 2M pages as discussed in Section 4.4. Since the performance of an application depends on many factors such as locality and caching, the reduction in DTLB misses does not correspond exactly with the improvement in performance. For the applications BT and FT the reduction in DTLB misses is lower, corresponding to a factor of 2-3. Correspondingly, the improvement in performance is lower. Because of limited kernel support on the Intel Xeons for OProfile, we could not measure these numbers there. We plan to include these numbers in the camera ready version of this paper.

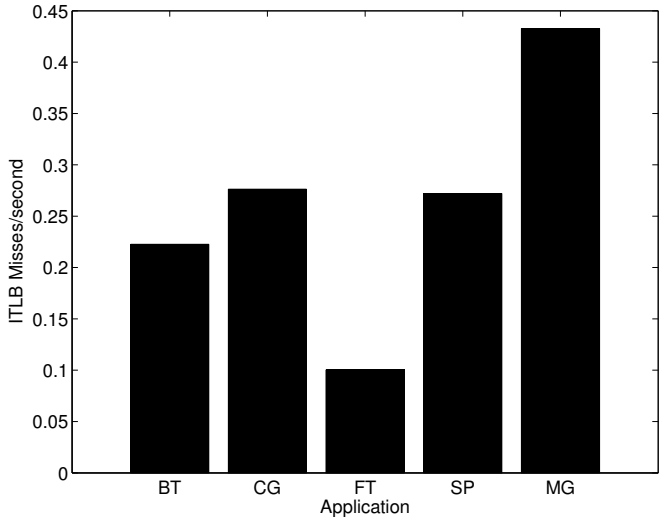


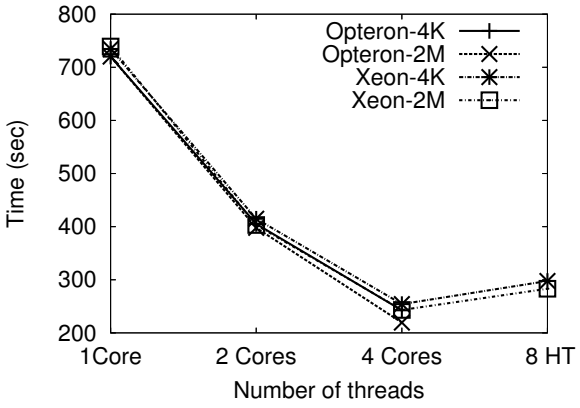
Fig. 3. Aggregate Instruction TLB misses per second of application run time with the application binary placed in 4K pages.

5 Related Work

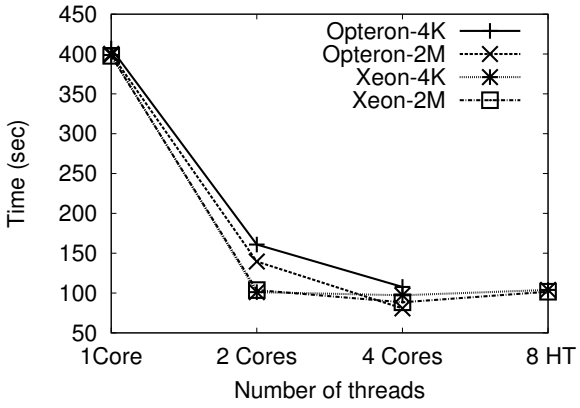
The work in this area can largely be categorized into investigations into the design of large page support for applications. The other category is evaluations of OpenMP primitives and applications on multicore architectures. We discuss each of these in detail.

Large Page Support for Applications: Cox et.al. proposed transparent operating system support for application memory using large page support. They considered a number of different design trade-offs while using a reservation based approach for allocating superpages of different sizes. Evaluation was in terms of a number of different sequential applications (including SP). Their finding showed that large pages can significantly reduce or even eliminate data TLB misses, and improve sequential application performance [21]. Our approach differs from theirs in that we allocate all the application data in large 2M pages at startup. Since parallel OpenMP applications on a multicore system are likely to have exclusive access to the system for the period of the run, this approach is practical and likely to yield a better improvement in performance. In addition, we evaluate the impact of large page support on the scalability of OpenMP parallel applications on modern multicore architectures with SMT. Other research directions [26, 28, 25] focused on the integration and design of huge pages. Performance evaluation focused on sequential applications. Other research did not consider applications in their performance evaluation [12, 15, 24]. Different TLB architectures for large pages were simulated and their impact on sequential application performance was evaluated in [27].

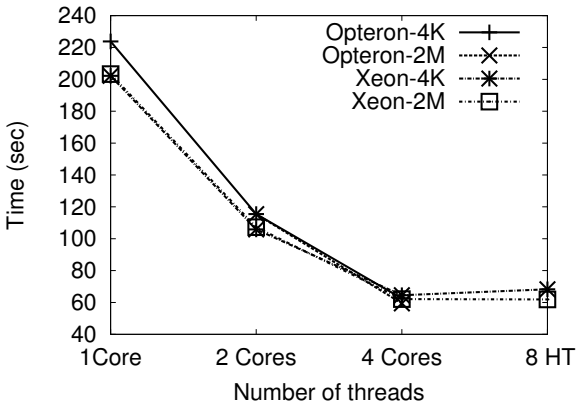
OpenMP and Multicore Architectures: There have been some recent investigations into evaluating the performance of OpenMP primitives and applications parallelized with OpenMP on multicore architectures. Chapman et.al. evaluated the EPCC, SPEC OMPN2001 and NAS Parallel (NPB) 3.0 on CMP and SMT architectures. We found similar conclusions regarding the scalability of parallel applications on the Xeon SMTs. Our works differs from their work in that we have evaluated the impact of large pages on the performance and scalability of NAS Parallel (NPB) 3.0 on CMP and SMT systems. Nikolopoulos



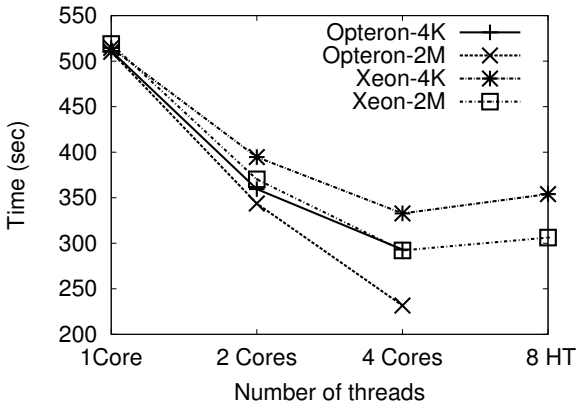
(a) BT



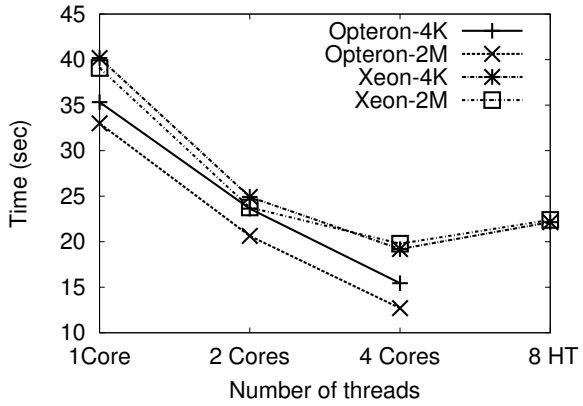
(b) CG



(c) FT



(d) SP



(e) MG

Fig. 4. Scalability of different applications on an Opteron and Intel Xeon platform with 4K and 2M pages. Single thread per core is used upto 4 threads. Two thread per core are used at eight threads (using hyperthreading on the Intel Xeon platform).

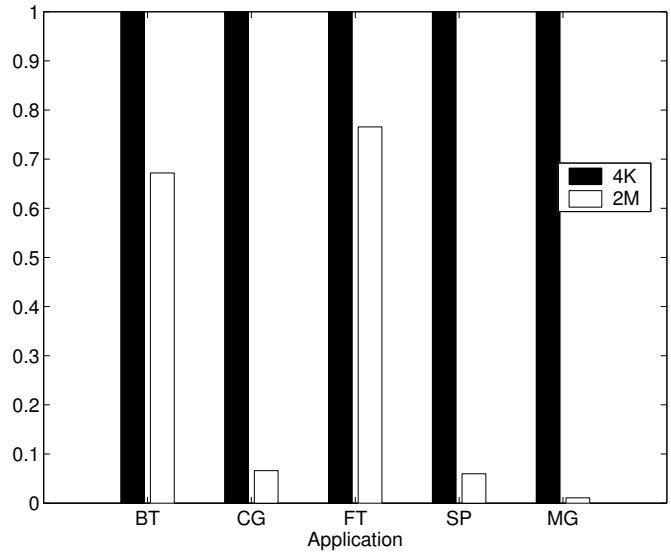


Fig. 5. Normalized Data TLB misses at 4 threads on an Opteron platform

et.al. [18] also evaluated the performance of NAS Parallel Benchmarks on SMT and CMP architectures. Their findings are similar to those of Chapman et.al. in that these benchmarks do not scale well on SMT architectures. In addition, they attempt to look at the results of TLB misses from the applications. Our work differs in that we evaluate the TLB misses and in addition, we also propose and evaluate the use of large or superpages to enhance the scalability of parallel OpenMP applications.

6 Conclusions and Future Work

In this paper, we have studied the impact of large page support available in modern processors on the performance of the OpenMP Parallel applications on a multicore Opteron and Intel Xeon platforms (with hyperthreading). We discuss the potential issues, design and evaluate an OpenMP system which uses large pages. Our evaluations show that the applications CG, MG and SP show an improvement of up to 25% at four threads on the Opteron platform using 2M pages instead of 4K pages. In addition, profiling tools show that the number of data TLB misses is dramatically reduced for these applications. In addition, the scalability of these applications is improved. 2M pages also help improve the performance on the Intel Xeon platform. Scalability on the Intel Xeon platform is also improved. However, because of the *pipeline flush* implementation of SMT on the Intel Xeons, the applications scale poorly when going from four threads to eight threads.

While 2M pages can improve performance of applications, transparent native kernel support for large pages is still not present in the Linux kernel. Ideally, the kernel and memory allocation library should be able to allocate a mix of large pages for the bigger allocation and the typical 4K pages for the smaller allocations. This will allow traditional applications to take advantage of large pages transparently. Finally, we would also like to evaluate the benefit of large pages on the performance of other programming paradigms such as MPI.

References

[1] Intel compiler 9.0 (c++ and fortran) for openmp. http://www.compunity.org/resources/compilers/intel/.

[2] Myricom, Inc. *www.myri.com*.

[3] Omni OpenMP Compiler Project. http://phase.hpcc.jp/Omni/.

[4] Omni/SCASH: Cluster-enabled Omni OpenMP on a software distributed shared memory system SCASH . http://phase.hpcc.jp/Omni/Omni-doc/omni-scash.html.

[5] OProfile-A System Profiler for Linux. http://oprofile.sourceforge.net/.

[6] The score system software. http://www.pccluster.org/.

[7] The GOMP Project. http://gcc.gnu.org/wiki/GOMP.

[8] The OpenMP Specification. http://www.openmp.org/.

[9] The Polaris Compiler. http://paramount.www.ecn.purdue.edu/ParaMount/Polaris/.

[10] Advanced Micro Devices, Inc. CPUID Specification. www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25481.pdf, 2006.

[11] Ananth Grama, Vipin Kumar, Anshul Gupta, George Karpis. Introduction to Parallel Computing. Addison-Wesley, 2003.

[12] N. Ganapathy and C. Schimmel. General purpose operating system support for multiple page sizes. In *USENIX Annual Technical Conference*, 1998.

[13] H. Jin, M. Frumkin and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. In *Technical Report NAS-99-011*, 1999.

[14] Intel Inc. Intel 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/design/processor/manuals/253666.pdf, 2006.

[15] Y. A. Khalidi, M. Talluri, M. N. Nelson, and D. Williams. Virtual memory support for multiple page sizes. In *Workshop on Workstation Operating Systems*, pages 104–109, 1993.

[16] Lei Huang, Chunhua Liao, Zhenying Liu and Barbara Chapman. An Evaluation of OpenMP on Current and Emerging Multithreaded/Multicore Processors. In *IWOMP*, 2005.

[17] Mark G. Sobell. HugeTLBFS: Translation Look-Aside Buffer Filesystem. In *A Practical Guide to Red Hat Linux: Fedora. Core and Red Hat Enterprise Linux, Second Edition*, 2004.

[18] Matthew Curtis-Maury, Xiaoning Ding, Christos Antonopoulos, Dimitrios Nikolopoulos. An Evaluation of OpenMP on Current and Emerging Multithreaded/Multicore Processors. In *IWOMP*, 2005.

[19] Mel Gorman. Understanding the Linux Virtual Memory Manager. Prentice Hall, 2004.

[20] Mitsuhisa Sato, Hiroshi Harada and Yutaka Ishikawa. OpenMP compiler for Software Distributed Shared Memory System SCASH. In *Workshop on Workshop on OpenMP Applications and Tool (WOMPAT)*, 2000.

[21] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. *SIGOPS Oper. Syst. Rev.*, 36(SI):89–104, 2002.

[22] OpenMP Architecture Review Board. OpenMP Application Program Interface. http://www.openmp.org/drupal/mp-documents/spec25.pdf, 2005.

[23] Richard McDougall and James Laudon. Multi-Core Microprocessors Are Here. In *;login: The USENIX Magazine*, volume 31, October 2006.

[24] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad. Reducing TLB and memory overhead using online superpage promotion. In *ISCA*, pages 176–187, 1995.

[25] I. Subramanian, C. Mather, K. Peterson, and B. Raghunath. Implementation of multiple pagesize support in HP-UX. In *USENIX*, 1998.

[26] M. Talluri and M. D. Hill. Surpassing the tlb performance of superpages with less operating system support. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 171–182, New York, NY, USA, 1994. ACM Press.

[27] M. Talluri, S. I. Kong, M. D. Hill, and D. A. Patterson. Tradeoffs in supporting two page sizes. In *ISCA*, pages 415–424, 1992.

[28] Zhen Fang and Lixin Zhang. A Comparison of Online Superpage Promotion Mechanisms. Technical Report UUCS-99-021, 1999.