Cluster Comput (2009) 12: 101–122
DOI 10.1007/s10586-009-0080-4

Paravirtualization effect on single- and multi-threaded memory-intensive linear algebra software

Lamia Youseff · Keith Seymour · Haihang You · Dmitrii Zagorodnov · Jack Dongarra · Rich Wolski

Received: 30 December 2008 / Accepted: 6 January 2009 / Published online: 24 January 2009

Abstract Previous studies have revealed that paravirtualization imposes minimal performance overhead on High Performance Computing (HPC) workloads, while exposing numerous benefits for this field. In this study, we are investigating the impact of paravirtualization on the performance of automatically-tuned software systems. We compare peak performance, performance degradation in constrained memory situations, performance degradation in multi-threaded applications, and inter-VM shared memory performance. For comparison purposes, we examine the proficiency of *ATLAS*, a quintessential example of an autotuning software system, in tuning the *BLAS* library routines for paravirtualized systems. Our results show that the combination of *ATLAS* and Xen paravirtualization delivers native execution performance and nearly identical memory hierarchy performance profiles in both single and multi-threaded scenarios. Furthermore, we show that it is possible to achieve memory sharing among OS instances at native speeds. These results expose new benefits to memory-intensive applications arising from the ability to slim down the guest OS without influencing the system performance. In addition, our findings support a novel and very attractive deployment scenario for computational science and engineering codes on virtual clusters and computational clouds.

Keywords Virtual machine monitors · Paravirtualization · AutoTuning · BLAS · High performance · Linear algebra · Cloud computing

1 Introduction

Virtualization has historically offered numerous benefits for high performance computing. It was, however ignored in computationally intensive settings as a result of its potential for performance retardation. A recent approach to virtualization, dubbed *paravirtualization* has emerged as a possible alternative. Originally developed to help consolidate servers in commercial settings, this OS-based approach to virtualization has grown rapidly in popularity. To provide the greatest flexibility, portability, and instrumentation possibilities, it comprises both software and hardware techniques. Originally explored by Denali [28] and Xen [6] in 2003, this technique in which the guest and the host OSs are both strategically modified to provide optimized performance is used today in several virtualization systems.

To this end, several studies [20, 29, 30] have measured the performance ramifications of running general HPC

This work is sponsored in part by NSF grants (ST-HEC-0444412 and CCF-0331645).

L. Youseff (✉) · D. Zagorodnov · R. Wolski
Dept. of Computer Science, University of California, Santa Barbara, USA
e-mail: lyouseff@cs.ucsb.edu

D. Zagorodnov
e-mail: dmitrii@cs.ucsb.edu

R. Wolski
e-mail: rich@cs.ucsb.edu

K. Seymour · H. You · J. Dongarra
Dept. of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, USA

K. Seymour
e-mail: seymour@eecs.utk.edu

H. You
e-mail: you@eecs.utk.edu

J. Dongarra
e-mail: dongarra@eecs.utk.edu

benchmarks on paravirtualized systems and they report near-native performance for different HPC workloads. At the same time, other studies have focused on the flexibility and functionality benefits of using modern virtualization techniques. For example, OS-level check-pointing [21], fault-tolerance and load balancing [9] are very attractive possibilities in HPC environments. In addition, other researchers [22, 25] have looked into dynamic adaptation of the guest OS for performance and application-customized guest OSs for scientific parallel codes [5, 16, 31].

Although previous studies of paravirtualization have addressed the general-case performance characteristics of HPC benchmarks, they lacked the investigation of the performance boundary and performance consequences under scarce memory conditions. This has a remarkable importance in HPC, because of the performance sensitivity of memory-intensive codes and autotuning Linear Algebra (*LA*) packages to the memory characteristics. In this vein, we provide a detailed study of the impact of the paravirtualized execution on *LA* codes that use autotuning software for portable performance. We focus on several areas in which paravirtualization may have potentially deleterious effects: VM interference with raw single-thread performance, interference between VMs in multi-threaded performance, and VM interference in shared memory communication efficiency.

Autotuning has become an important technology for “core” libraries that are performance-critical since these systems often require complex and otherwise labor-intensive configuration to achieve maximal performance levels. Thus, our goals are two fold. First, we wish to understand how autotuning is affected by paravirtualization. For example, we wish to know whether the autotuning software can “sense” the presence of paravirtualization during the tuning process. Secondly we wish to explore the potential impact paravirtualization may have on highly tuned numerical software. While it may be that a vanilla installation is unaffected as has been shown in previous studies [30], in this work we investigate the effects of paravirtualization on the performance boundary.

In particular, we study the efficacy of *A*utomatically *T*uned *L*inear *A*lgebra *S*oftware (*ATLAS*) in detecting system configurations for paravirtualized systems. We then use DGEMM matrix-matrix multiplication as a memory-intensive code to measure performance in Mflops for double precision arithmetic and compare the performance of several OS kernels with varying main memory allocations. With this in mind, we investigate the different attributes of the memory hierarchy of the paravirtualized system.

The main contribution of this paper is an exposition of how paravirtualization (as implemented by Xen) impacts the performance of *LA* codes, particularly with respect to its use of the memory hierarchy. Also, with the increasing prevalence of multi-core architectures, the issues of memory contention as well as efficient communication and synchronization between cores will be crucial to large-scale application performance. Thus, we have studied whether running highly tuned *LA* code in different VMs causes more performance perturbation than multiple threads within a single VM. Finally, as the ever increasing number of cores per chip makes evident the need for low-overhead communication, we study the efficiency of using shared memory between VM instances on different cores. Our findings complete along with our previous studies [29, 30], the investigation of the feasibility of utilizing clusters of virtual machines to execute scientific codes, without sacrificing performance. Our research, therefore advocates the huge potential of deploying scientific codes on virtual clusters and computing clouds. In turn, this presents new and very attractive deployment scenarios for computational sciences and engineering codes. The novel deployment scenarios are not the only appealing implication of our research, but the saving on the computing expenditure can be another very desirable advantage. As the cost of virtual clusters comprises a fraction of computing hardware acquisition and maintenance costs, our results have the potential to influence the total cost of the inquiry process in the computational sciences and engineering.

The paper is structured as follows. We present a short survey on paravirtualization in the next section, as well as the terminology we will use. We detail our experimental settings in the following section. Section 4 presents the impact of paravirtualization on *ATLAS* system detection and the performance of the generated and hand-tuned routines. Next, Sect. 5 investigates the effect of the paravirtualization on the memory hierarchy of the system, by describing its impact on a memory intensive dense matrix-matrix multiplication performance and characterizing its RSS (Resident Set Size), swap and TLB activity. Section 6 continues the examination of matrix multiplication to determine whether paravirtualization has a negative effect on the performance of multiple simultaneous instances. Section 7 discusses the impact of paravirtualization on shared memory performance. Finally, we discuss the implications of our work in Sect. 8 and present our conclusions in Sect. 9.

2 Background

Historically, virtualization has been a technique for providing secure and flexible hosting of user and system programs spanning several trust levels. In an Internet computing setting, this hosting capability has relied on language semantics (e.g. Java) to permit the importation of untrusted code for execution by a software virtual machine. While the approach has been successful, incompatibilities between the

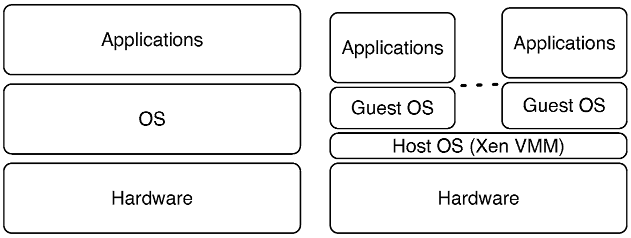


Fig. 1 The two software stacks in our experimentation settings. (i) The stack on the *left* is the traditional (native) software stack. (ii) The stack on the *right* shows the virtualized software stack

virtual machine mandated by the language and the typical organization of actual computational resources (processors, memory, I/O subsystems, etc.) imposes a performance penalty on virtualized program execution. Many powerful and elegant optimization techniques have been developed to minimize this penalty, but at present language virtualized systems still do not achieve native execution speeds for numerically-intensive applications.

A recent approach to virtualization, dubbed *paravirtualization* has emerged as a possible alternative. Paravirtualization is a software virtualization technique which allows the virtual machines to achieve near native performance. In paravirtualized systems, for example Xen [6], the system software stack is augmented, as illustrated in Fig. 1. The stack on the left shows the traditional OS deployment stack, while the right stack portrays the paravirtualized deployment stack. In the latter, the hypervisor[1] occupies a small part of the main memory, and acts as a moderator layer between the hardware and the guest OS kernels. On top of the hypervisor, two types of guest OSs are run. The first type is regarded as a privileged services guest OS, which provides OS services to the other less-privileged OS and has more direct access to memory, devices, and the hardware in general. There must be at least one privileged virtual machine per physical machine. The other kind of guest OS is a less privileged OS kernel, which uses paravirtualized device drivers and has moderated access to the hardware. The privileged guest OS is responsible for running virtualization software tools that manage, start, monitor and even migrate the other less-privileged domains.

In order to measure the performance impact of paravirtualization on autotuning software, we used *A*utomatically *T*uned *L*inear *A*lgebra *S*oftware (*ATLAS*) [10, 27]. *ATLAS* focuses on applying empirical search techniques to provide highly tunable performance for linear algebra libraries. It empirically explores the search spaces for the values of the different parameters for *B*asic *L*inear *A*lgebra *S*ubprograms (*BLAS*) [8, 11, 18] and *L*inear *A*lgebra *Pack*age (*LAPACK*) [3] routines for matrix operations. Those kinds of matrix kernels are among the most widely studied and optimized routines in computational science due to their influence on the overall performance of many applications. Traditionally, developers either carefully optimized these algorithms by hand or they relied on compiler optimizations to improve the performance. Hand-tuning requires a lot of expertise and quite a bit of effort from the developer. Even if the hand-tuning is successful, it is not often portable to other architectures, so the developer has to repeat the process multiple times to support a useful set of platforms. On the other hand, using compiler optimizations requires almost no effort from the developer, but may only give modest results, especially compared to the theoretical peak performance of the machine. Many compiler optimization techniques such as loop blocking, loop unrolling, loop permutation, fusion and distribution have been developed to transform programs written in high-level languages to run efficiently on modern architectures [1, 23]. Commonly referred to as model-driven optimization, most compilers select the optimization parameters such as block size, unrolling factor, and loop order with analytical models. The models may be based on real architectural attributes or other heuristics, but compilers are burdened with the task of handling a wide variety of code, so the built-in optimizers usually cannot compete with experienced hand-tuners. In contrast, empirical optimization techniques generate a large number of code variants for a particular algorithm (e.g. matrix multiplication) using different optimization parameter values or different code transformations. All these candidates run on the target machine and the one that gives the best performance is picked. Using this empirical optimization approach, projects like *ATLAS*, *PHiPAC* [7], *OSKI* [26], and *FFTW* [13] can successfully generate highly optimized libraries for dense and sparse linear algebra kernels and FFT respectively. This empirical approach has been recognized as an alternative approach to traditional compiler optimizations and machine specific hand-tuning of linear algebra libraries, since it normally generates faster libraries than the other approaches and can adapt to many different machine architectures. In a recent report from Berkeley on the future of parallel computing [4], software autotuners were regarded as a way to enable efficient optimizations and should become more adopted in translating parallel programs and code generation. Towards this end, we expect autotuners will be more embraced in the near future, and will run on virtualized machines such as the computing clouds. Hence, we are investigating in this paper the impact of paravirtualization on the operation of autotuners.

With this in mind, we used the performance and the parameter values of the autotuned *BLAS* library as an indication of the efficiency of the autotuning process in paravirtualized

[1] The hypervisor is a small piece of software that runs directly on the hardware and acts as a slim layer between the guest OSs and the hardware. It is also referred to as virtual machine monitor (VMM). Accordingly, the OS kernels that run on the hypervisor are termed virtual machines, or guest OSs.

Table 1 Mathematical notations for the routines in *BLAS* and *LAPACK* libraries

Routine	Operation	Description
GEmmNN	$C = \alpha AB + \beta C$	General Dense non-transpose non-transpose Matrix-Matrix Multiplication
GEmmNT	$C = \alpha AB^T + \beta C$	General Dense non-transpose transpose Matrix-Matrix Multiplication
GEmmTN	$C = \alpha A^T B + \beta C$	General Dense transpose non-transpose Matrix-Matrix Multiplication
GEmmTT	$C = \alpha A^T B^T + \beta C$	General Dense transpose transpose Matrix-Matrix Multiplication
GEmvN	$y = \alpha Ax + \beta y$	General Vector-Matrix Multiplication
GEmvT	$y = \alpha A^T x + \beta y$	General Vector-Matrix transpose Multiplication
GER	$A = \alpha xy^T + A$	General Rank one update

Table 2 Description of the parameters used in tuning the *BLAS* routines

Parameter	Description
N_B	L1 Blocking factor
ma	MULADD boolean flag to indicate whether the MULADD is done as one operation or not
la	Latency between floating point operations
nb	Blocking factor used in each specific routine
nu, mu and ku	Unrolling factors for M, N, K loops respectively
X unroll and Y unroll	Unrolling factors for X, Y loops respectively

environments. *ATLAS* is convenient to use for these experiments because of its widespread use for generating tuned *LA* libraries. In addition, the detected characteristics of the system can be easily examined in the log files and compared among the different OS kernel configurations. Also, since *ATLAS* typically achieves 75–90% of peak performance in the native configuration, it should give a good indication of whether the various OS kernel configurations are capable of high sustained floating point performance.

Notice that *ATLAS* essentially performs a “parameter-sweep” search of the performance space so that it can identify the values of the specific parameters that yield the best performance (among those tested). The resulting library configuration typically achieves a better performance than a generic installation. Because applications running near peak machine speed can be more performance sensitive to effects introduced by their OS environment (e.g. OS noise), we wish to examine the degree to which paravirtualization interferes with an optimized installation. Notice also that the set of parameters identified by *ATLAS* are conveniently logged making it possible to use them to detect specific performance differences between native and paravirtualized execution. That is, by comparing *ATLAS* tuning logs for native and virtualized optimization, we should be able to identify immediately how paravirtualization is affecting the execution of optimized *LAPACK* libraries.

To help understand some of the parameters and the subroutine names mentioned in this paper, we will briefly describe the *BLAS* and *LAPACK* naming convention (for full details, see [3]). Subroutines are named `XYYZZ` or `XYYZZZ`, where `X` indicates the data type (`S` for single precision, `D` for double precision, etc.), `YY` indicates the matrix type (`GE` for general, `GB` for banded, etc.), and the final `ZZ` or `ZZZ` indicates the computation performed (`MM` for matrix-matrix multiply, `MV` for matrix-vector multiply, etc.). Therefore, `DGEMM` would be a double-precision general matrix-matrix multiplication. In this matrix-matrix multiplication routine, an $M \times K$ matrix A multiplies a $K \times N$ matrix B, resulting in the $M \times N$ matrix C. *ATLAS* finds the optimal value for the best blocking and loop unrolling factors for on-chip multiply using timings, i.e. it examines the search space by trying different values for blocking and loop unrolling. In Table 1, we outline the *BLAS* routines that *ATLAS* optimizes and in Table 2, brief descriptions of the different parameters for the routines are outlined.

3 Experimental settings

We ran our experiments on a Pentium D dual-core machine, where each core is a 2.8-GHz Pentium with an 800-MHz processor bus, 16 KB of L1 cache and 1024 KB L2 cache. The machine memory system uses a 533-MHz bus with 1 GB of dual interleaved DDR2 SDRAM memory.

In order to find the performance ramifications of paravirtualization, we compare the performance of three types of

OS kernels. Furthermore, we test two configurations that differ in the main memory size allocated at boot time for each OS kernel. The first kernel is a Fedora Core Linux 2.6.19 kernel, which we used as a base performance kernel, and henceforth referred to as "native". For this kernel, the device drivers, applications and *BLAS* libraries run directly in the OS (without virtualization), which is the common software stack used nowadays in HPC clusters.

On the other hand, the paravirtualized software stack is different, as we described in the previous section. We used Xen as our paravirtualizing software, with the hypervisor in the first 32 MB of the main memory. Furthermore, in Xen terminology, the privileged guest OS is dubbed *Dom*0 (for *Dom*ain 0) while the less privileged guest OS is dubbed *DomU* (for *Dom*ain *U*nprivileged). We adopt this terminology for the rest of our paper. For each of the three OS kernels (native, *Dom*0, *DomU*), we test the performance with two main memory configurations: 256 MB and 756 MB. The reason for changing the total memory assigned to the systems was to test the performance of the system under limited memory conditions and to generate near-boundary memory cases for virtualized systems. We disabled the balloon driver[2] in Xen domains in order to isolate the impact of the balloon driver on memory performance and to build a fair comparison between the different systems. In our experimentation, we used Linux kernel 2.6.16 as the guest OS for both *dom*0 and *domU*, patched with Xen 3.0.4. All the OS kernels were built with SMP support.

We use *ATLAS* 3.7.35, the latest version available for autotuning the *BLAS* routines. We compare the performance achieved using the *ATLAS*-generated code (with and without SSE2 support). We also compare the performance achieved by the DGEMM routine for different matrix sizes. In addition, threading was enabled in all *ATLAS* builds to allow *ATLAS* to build parallel libraries.

4 Autotuning software systems

ATLAS is one of the earliest autotuning software systems for performance optimization. We use *ATLAS* in our research as a quintessential example of an autotuner for processors with deep memory hierarchies and pipelined functional units. As we described in Sect. 2, *ATLAS* uses an empirical search methodology to optimize the different routines for *BLAS* and *LAPACK*. This search process is composed of three key phases. In the first phase, *ATLAS* focuses on detecting the system characteristics. Through a probe process, *ATLAS* collects information about cache size of the system, the floating point unit (FPU), the number of registers, and other architectural information. The second phase is concerned with determining the best values of parameters to be used in generating the *BLAS* and *LAPACK* routines based on the detected system characteristics and the results of the empirical search. After tuning, *ATLAS* runs cache benchmarks to determine the optimal value for *CacheEdge*, which represents the value of the cache size for blocking the matrix multiplication routines. Finally, *ATLAS* uses all the information it gathered to generate the optimized *BLAS* library. In the next three subsections, we detail the performance of *ATLAS* in each of the three phases respectively. For those results, we found that the precision of the multiplication (i.e. single versus double) does not impact the difference in performance between the OS kernels. Therefore, we only detail the double precision performance results.

4.1 System characteristics detection

In order for *ATLAS* to autotune the *BLAS* libraries, it starts its operation by probing the system characteristics. Table 3 shows the output of *ATLAS* for each of these parameters. The first row shows that *ATLAS* detected L1 cache to be of size 16 KB for all the OS kernels. The second row illustrates the number of registers detected in each of the systems, for which *ATLAS* detected 7 registers for all three OS kernels. The length of the floating point pipeline (in cycles) is presented in the third row, while the number of FPU registers is presented in the fourth row.

Furthermore, Fig. 2 shows the floating point unit (FPU) register-to-register performance (i.e., with no memory latency) as measured by *ATLAS*. For each of these performance numbers, we present an average of 20 runs with error bars reflecting the margin of error for a 95% confidence level of the mean. In this figure, the Y-axis represents the performance in Mflops. Therefore, we concluded from the measurements that there is no significant performance difference between the OS kernels for FPU operation. Overall, we concluded from these results that the paravirtualization does not alter the system characterization, nor does it impose any performance overhead in register-to-register performance for floating point operations.

4.2 Cache blocking size configuration

Tuning the *CacheEdge* (i.e., cache blocking parameter) can help increase performance and reduce the memory usage of *BLAS* routines. In this phase, *ATLAS* attempts to determine the optimal cache size for blocking the matrix-matrix multiplication routines. It first tests the blocking performance using only L1 cache, then uses different values of L2 cache.

We compared the performance achieved by each OS kernel for L1 cache and each value of L2 blocking. Figure 3

[2] The balloon driver in Xen allows the domains to grow and shrink dynamically in their total main memory allocation, according to their runtime memory workloads.

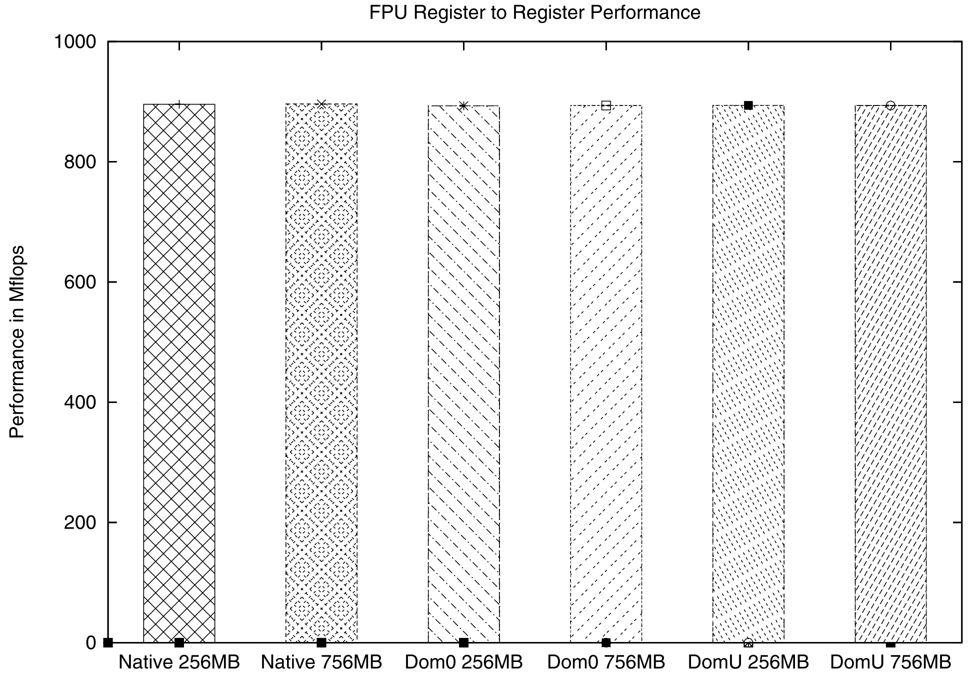


Fig. 2 Performance of the register-to-register FPU for double precision numbers; as detected by *ATLAS* for the OS kernels

Table 3 System characterization as detected by *ATLAS* for the OS kernels

	Native_SMP		Dom0_SMP		DomU_SMP	
Parameter	256 MB	756 MB	256 MB	756 MB	255 MB	756 MB
L1 Cache Size	16 KB	16 KB	16 KB	16 KB	16 KB	16 KB
Sys Info nreg	7	7	7	7	7	7
FPU: Pipeline cycle	6	6	6	6	6	6
FPU: Registers num.	15	15	15	15	15	15

depicts the performance in Mflops of a double precision matrix-matrix multiplication of dimension 2500 using only L1 cache blocking, while Fig. 4 represents the performance of using L2 blocking. All the numbers reported here are the average of 20 runs. For the latter figure, the x-axis represents the size of L2 cache in KB used in blocking, while the y-axis represents the corresponding performance. The error bars reflect the margin of error for 95% confidence level. Note that we extended the *ATLAS* subprogram which does the CacheEdge measurements to evaluate the performance beyond the physical L2 cache size in order to monitor any difference. However, no performance difference was detected between the native and paravirtualized kernels. Figure 5 shows a histogram of the final CacheEdge selected by *ATLAS* for the 20 runs, after disregarding the runs where *ATLAS* chose only L1 blocking. The reason *ATLAS* does not choose the same CacheEdge size for L2 blocking every time is that the code is sensitive to the slight performance difference for cache sizes between 512 KB and 2048 KB. Therefore, a small variability in the performance impacts the chosen CacheEdge but does not impact the overall performance as Figs. 3 and 4 show. That is, a small difference in performance will cause *ATLAS* to choose a different power of 2 for the cache block size (a relatively large change). The histogram in Fig. 5 reflects the variation of cache-block size which *ATLAS* selected over different runs, but Figs. 3 and 4 show that this variation does not ultimately affect the overall performance (note the small error bars in the figures).

In addition, Table 4 outlines the median values of the CacheEdge selected. The reason we chose to report the median rather than the mean is that *ATLAS* chooses among different categorical values of L2, i.e., the median was more representative of the optimal CacheEdge's choice. From Table 4 and Fig. 5, we gather that the selection of the optimal CacheEdge performance is similar for all the OS kernels. This shows that *ATLAS* finds minimal or no difference between the different OS kernels in choosing their optimal L2 blocking size.

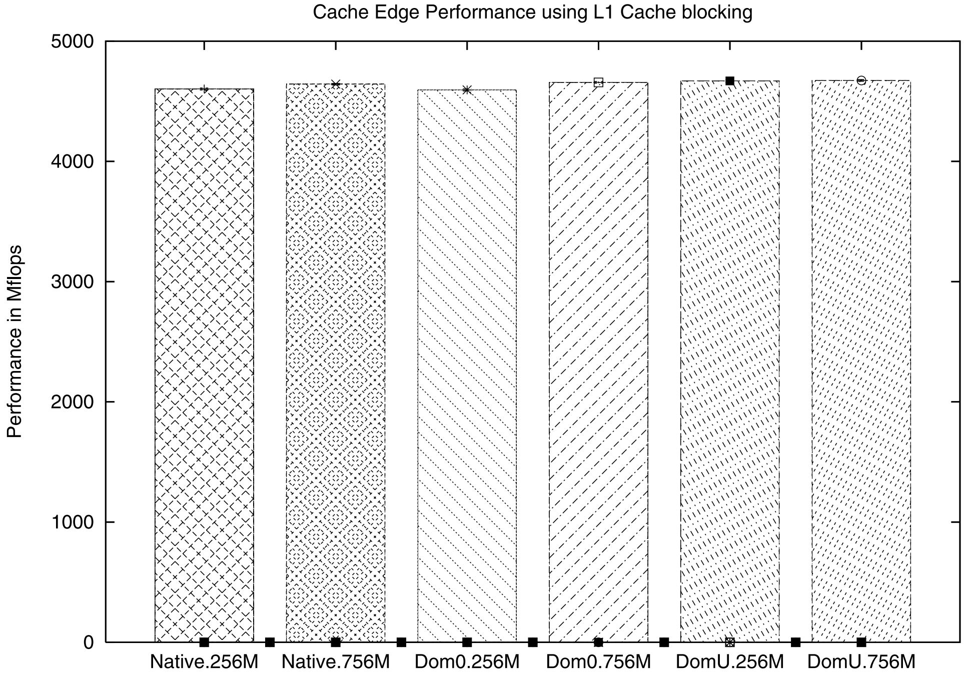


Fig. 3 Performance of the 2500d matrix-matrix multiplication for L1 cache blocking

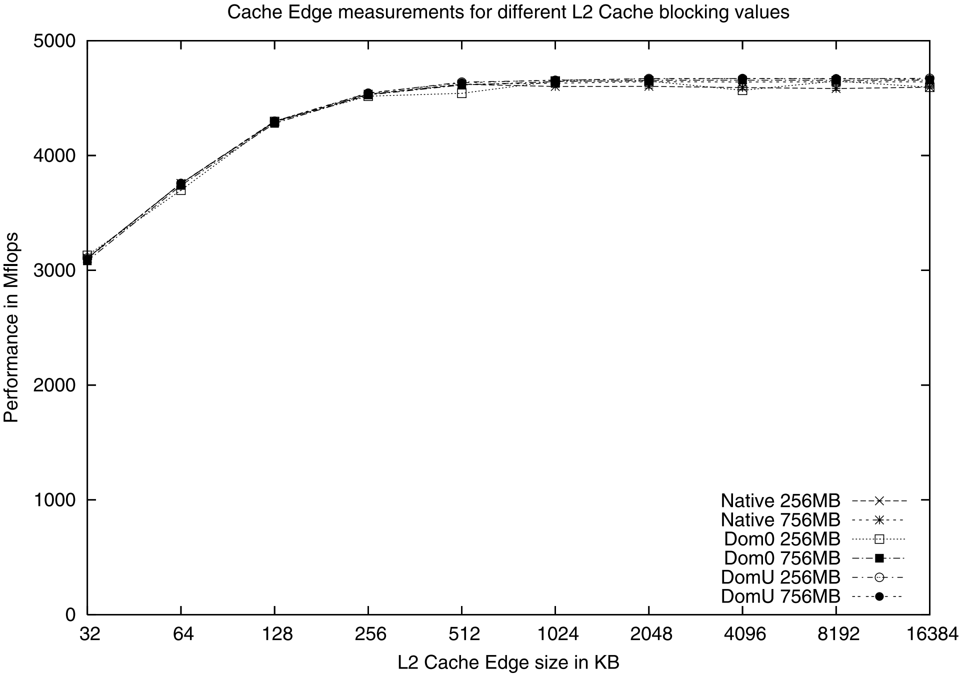


Fig. 4 Performance of the 2500d matrix-matrix multiplication for L2 cache blocking

4.3 Routines generation and tuning

In order for *ATLAS* to obtain the best performance from the system, it runs different routines and measures their performance to choose the most efficient optimization for the *BLAS* library customization. Some of the computational kernels that come with *ATLAS* are hand-written assembly routines, while others are autogenerated based on the output of the system probe phase. In many cases and especially for the popular architectures, the hand-written computational kernels perform much better than the generated routines, since the former kernels make use of specific special architectural features that *ATLAS* code generator does not currently support. In this section, we compare the performance of the *ATLAS* generated codes and the hand-written codes for the different OS kernels.

Fig. 5 Histogram of the CacheEdge selected by *ATLAS*

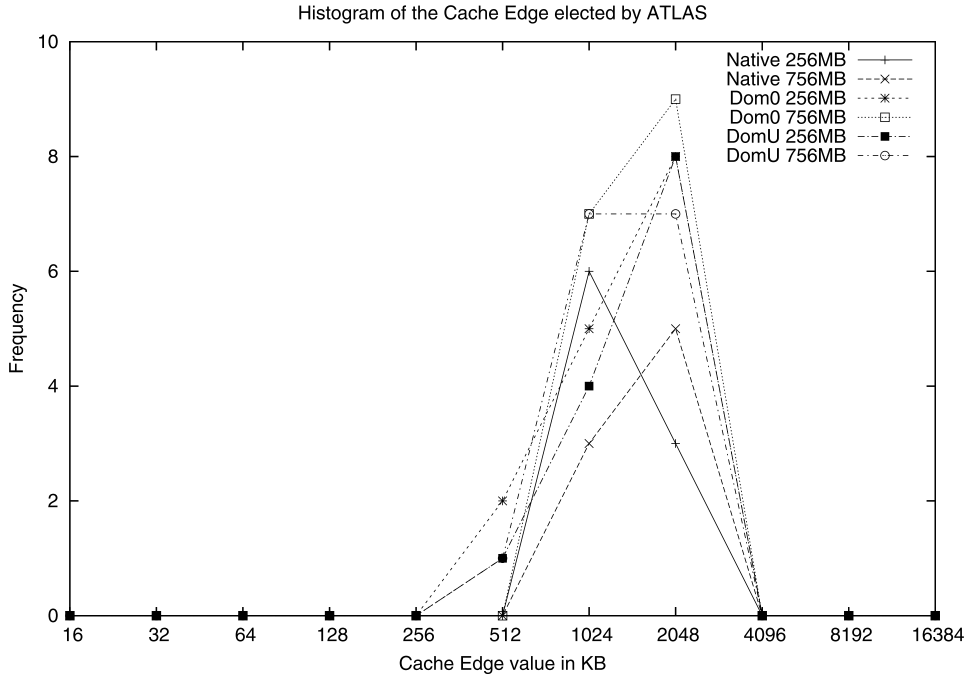
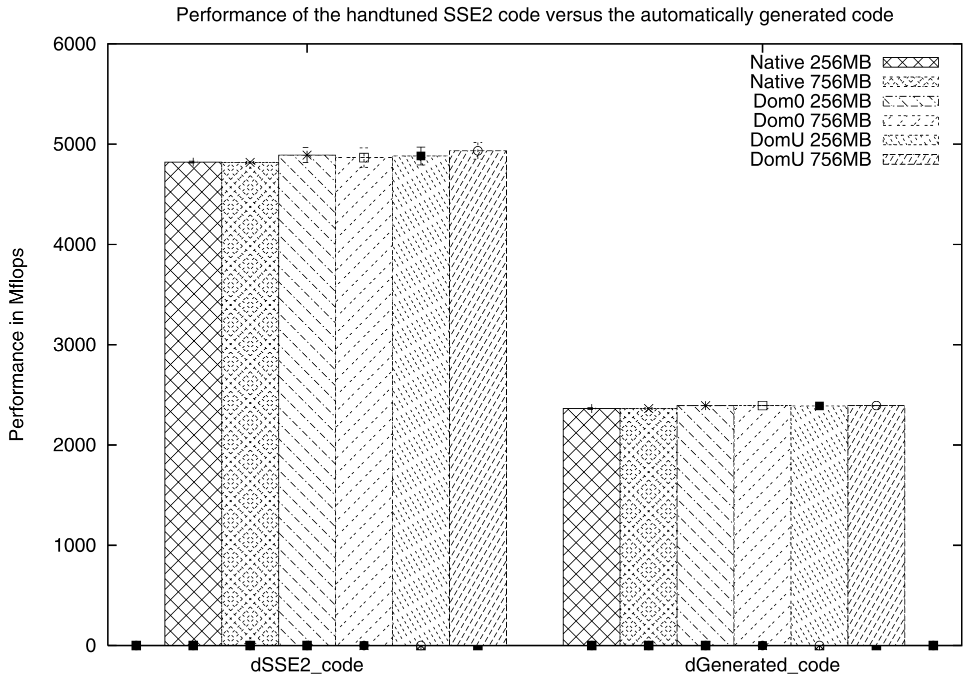


Fig. 6 Performance in Mflops of double precision matrix-matrix multiplication for the OS kernels, using the handwritten, and autogenerated routines



We found that, for all the routines, *ATLAS* selected the handwritten versions as they performed about 4× faster than the generated code for single precision and 2× faster for double precision. Figure 6 shows the performance achieved by the handwritten routines in comparison to the performance of the generated routines for the different OS kernels for double precision. The reason behind the better performance of the hand-written code is its use of the SSE2 assembly instructions, which run on the SSE2 unit in the Pentium CPUs. SSE is an extension to the streaming SIMD (single instruction, multiple data), which is a recent addition to the Pentium processors to support floating point operations and is backed with an extra set of instructions for SIMD on x86. The handwritten code, which includes the SSE2 (i.e., second version of SSE) assembly instructions, made use of this extra processing power. SSE2 allows the processor to perform

Table 4 Median of the CacheEdge selected by *ATLAS* for the OS kernels

OS kernels	Selected CacheEdge
Native 256 MB	1024 KB
Native 756 MB	1536 KB
*Dom*0 256 MB	1152 KB
*Dom*0 756 MB	1536 KB
DomU 256 MB	1536 KB
DomU 756 MB	1024 KB

Table 5 *ATLAS* automatically chose the same values for each tuning parameter, irrespective of the underlying OS kernel or its main memory allocation

Routine	Parameter	Value
	N_B	36
GEmmNN &	*ma*	1
GEmmNT &	*lat*	6
GEmmTN &	*nb*	48
GEmmTT	*mu*	6
	nu	1
	ku	48
GEMVN	Xunroll	32
	Yunroll	4
GEMVT	Xunroll	2
	Yunroll	16
GER	*mu*	16
	nu	1

two and four times the number of floating point operations per cycle in double precision and single precision, respectively.

In order to understand the difference in the performance of the autogenerated codes, we examined the values of the parameters used by *ATLAS* to generate the distinct routines, as well as the performance of these routines. For completeness, Table 5 illustrates the values for the different parameters that *ATLAS* selected to optimize the computational routines. Each of these factors is optimized for each of the computational routines (shown in the first column in the table) in *BLAS* and *LAPACK* libraries. The routine names are described in Sect. 2, and the average performance over 20 runs is shown in Fig. 7.

After *ATLAS* has picked the optimal values for the tuning parameters, it generates the computational routines for the *BLAS* libraries using those values. We investigated the performance of the generated routines using the tuned parameters. Figure 7 represents the performance of the generated routines for the different kernels, in double precision. In this figure, the Y-axis represents the performance in Mflops. We noticed again that there was no significant difference in the values of the tuned parameters or the performance of the autogenerated routines between the different OS kernels. In conclusion, we detected no difference between *ATLAS* system detection, and auto-tuning between the native and paravirtualized OS kernels.

5 Memory intensive applications

In the second set of experiments, we are investigating the impact of paravirtualization on the different levels of the memory hierarchy. Towards this end, we explore the paravirtualized memory hierarchy behavior using a memory-intensive application. For that, we use double-precision matrix multiplication code that uses the *BLAS* level-3 libraries as a driver code for our experiments. This driver code is characterized by a growing memory consumption of up to 350 MB. We describe our DGEMM driver code in the next subsection, and the memory hierarchy performance in the subsequent subsections.

5.1 DGEMM driver description and performance

Among the other *BLAS* library routines, *ATLAS* tunes the DGEMM routine to efficiently execute double precision matrix-matrix multiplication at optimal performance. We generated a driver code that uses the *ATLAS*-optimized DGEMM routine for square matrix sizes ranging from 100 to 4000 and we recorded the time and performance in Mflops attained for each matrix size. The general pseudocode of the driver is outlined below.

```
for Dimension 100 to 4000, step 100
 for runs 1 to n, step 1
  malloc matrices A, B, C;
  randomize values in A, B, C;
  t1 = time;
  call dgemm (A, B, C, alpha, beta);
  t = time - t1 ;
  free A, B, C;
 calculate the MFLOPS of runs;
```

For each matrix dimension, we rerun the same driver code at least 5 times. In addition, we ensured that the OS kernel is not penalizing our driver code because of its increasing memory or computational requirements, by forking a different process for each run for each matrix dimension. As outlined in the DGEMM driver pseudocode, the driver code creates 3 matrices, and fills them with random numbers, causing the page table entries for this newly allocated memory to be populated and avoiding having Linux use the lazy memory allocation scheme.[3] We measure the time

[3] Recent Linux kernels use an optimization scheme dubbed *lazy-malloc*, which delays the actual allocation of the memory requested by the process until its actual use.

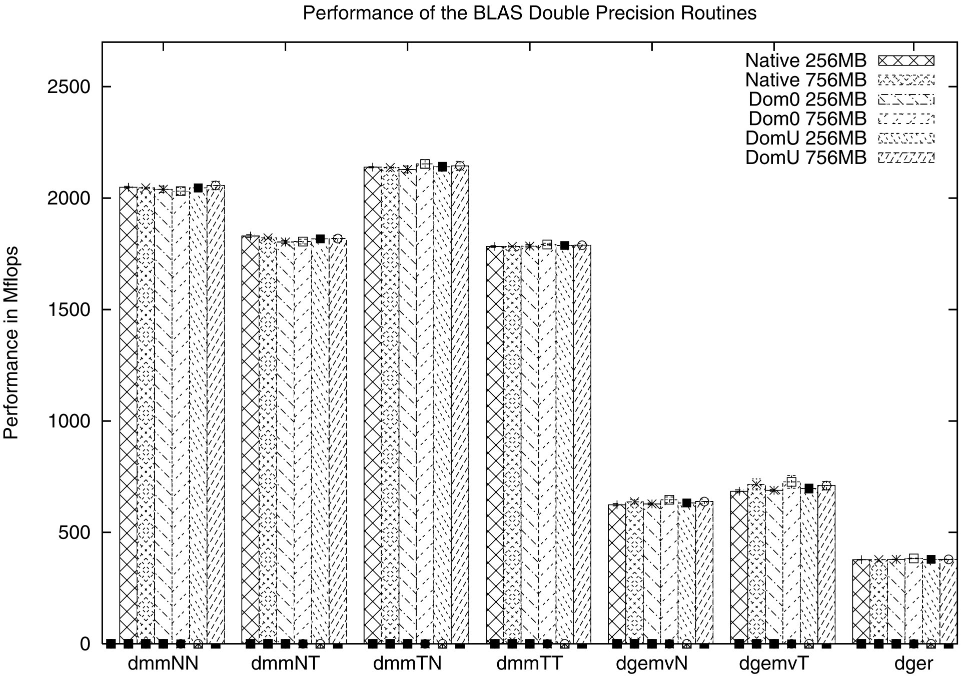


Fig. 7 Performance of the *ATLAS*-generated *BLAS* routines based on the tuned parameters for double precision

needed by the DGEMM code to calculate the multiplication output, and free the allocated memory afterward. Therefore, the memory requirement by the driver code process for a specific matrix dimension x is $3x^2$. Consequently, the data-section memory requirement of the driver code ranges from 237 KB to 366 MB for matrix dimension from 100 to 4000, respectively.

We show the results of our experiments in Fig. 8, which consists of 6 subfigures, one for each of the OS kernel configurations. The upper row of subfigures represents the performance curves of the native, *Dom*0 and *DomU* kernels configured with 256 MB main memory, while the lower row of subfigures represents the performance curves of the same kernels, but with 756 MB main memory. Each of the six subfigures plots the performance in Mflops on the Y-axis as a function of the matrix dimension on the X-axis. The curves show the average Mflops attained by the driver code at each matrix dimension, while the error bars demonstrate the minimum and the maximum Mflops attainable through the execution of several runs of the driver code.

For all kernel configurations in the 6 subfigures, the performance increases until it exceeds 4 Gflops at matrix dimension of 700, which corresponds to having DGEMM driver process memory size of 11.21 MB. The reason behind the increasing Mflops between matrix dimensions of 100 to 700 is the resolution of the timing. The total execution time of the DGEMM process is short for small size matrices, which takes less than 0.25 seconds, and therefore the problem size is not large enough to achieve the peak Mflops of the machine. Nevertheless, the growing curve was similar for all of the OS kernel configurations up to matrix dimension 700, whether their total main memory was 256 MB or 756 MB. The performance of the driver code, afterward stabilizes up to dimension of 3100 for all kernels equally.

Although the main memory size did not impact the Mflops achieved by the DGEMM driver code up to matrix dimension of 3100, its impact was clearly encountered by larger matrices running on limited main-memory OS kernels. For the three OS kernels with the main memory allocation of 756 MB (lower row of figures), the performance attainable by the driver code stabilized for matrix dimensions larger than 800. On the other hand, the three other OS kernels with main-memory allocation of 256 MB experienced performance degradation when the DGEMM driver memory requirement exceeded a certain threshold. This threshold was the matrix dimension of 3100 for all of the kernels, which corresponds to the matrices memory size of 219 MB. The cause of this performance degradation is the memory swapping of the driver memory, as we show in the next subsection. Because the total main memory is limited to 256 MB in these three OS configurations, the OS kernel starts to swap out the DGEMM driver data as its size increases, and as physical memory becomes constrained. The OS kernels with 756 MB do not experience the same performance degradation, since their main memory allocations are much larger than the DGEMM memory consumption.

In order to thoroughly understand the memory consumption in each OS kernel configuration and its impact on the DGEMM driver process, we had to study it in the context of other entities using the memory, including the data and code sections of the OS kernels.

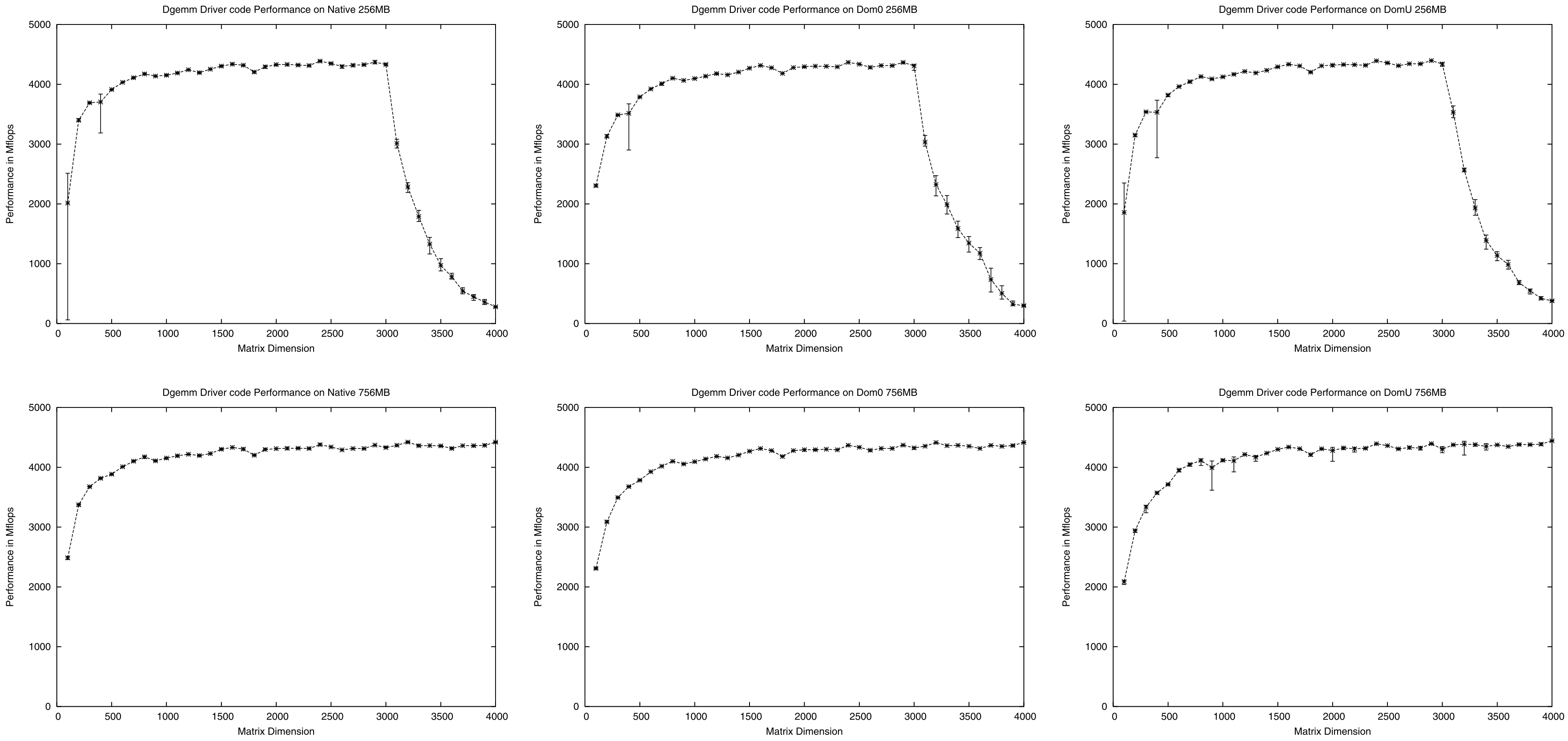


Fig. 8 The performance in Mflops for the DGEMM driver code as a function of the matrix dimensions. The *upper row* of figures portrays the performance for the kernels with 256 MB memory, while the *lower row* of figures represents the performance for the kernels with 756 MB

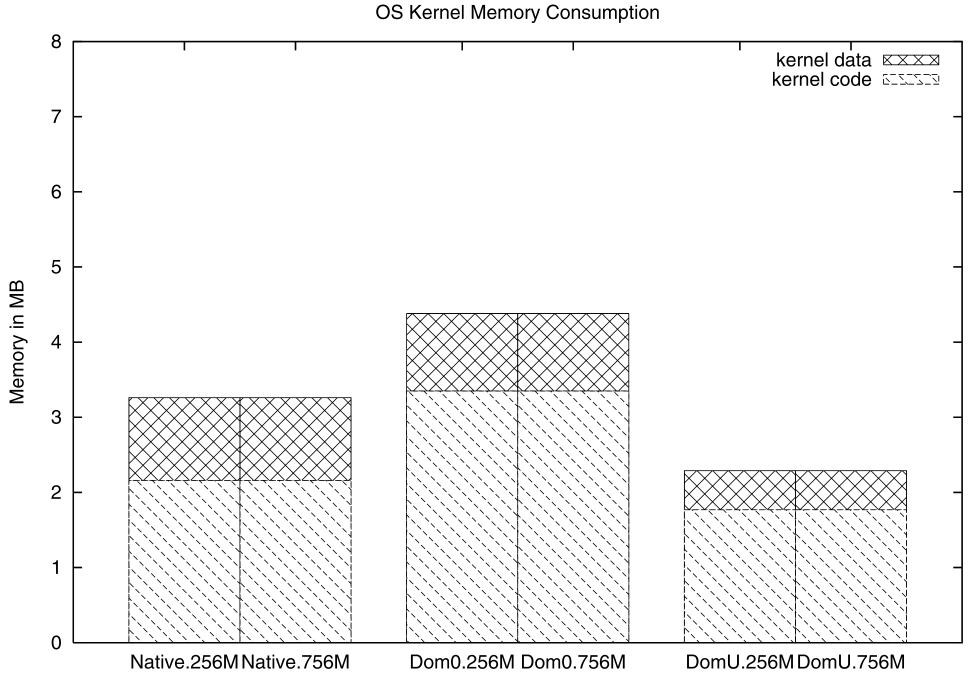


Fig. 9 Memory consumption of the OS kernels with different memory configurations

5.2 Kernel memory and DGEMM resident set size

The first portion of memory we investigated was the kernel memory. Figure 9 is a stack bar graph of the kernel memory components and how much space they occupy in the physical RAM. The kernel memory is consumed by two main components: the kernel code, and the kernel data. *DomU* has the smallest kernel code size of the three OS kernels, which is a byproduct of the exclusion of all the hardware driver codes from the unprivileged domain. Xen has implemented a split driver architecture, where it splits the driver interface into two separate portions with a well-defined API between them. *Dom*0 contains all of the physical drivers to the hardware, as well as the back-end of the split driver codes, while *domU*s contain only the front-end drivers, which are much smaller in code size, and much simpler in interfaces. As a result of this split design, *DomU* has the smallest kernel code size, while *Dom*0 has the largest.

The second component of the kernel memory is the kernel data, which is the amount of memory statically reserved for the kernel data structures. For the native kernel and *Dom*0, they both have almost the same size of data in memory (1.05 MB), while the *DomU* kernel has half much this data memory (0.52 MB). The reason behind that is also the split structure architecture. Since the interface between the back-end and front-end drivers is simpler, *DomU* is keeping much simpler data structures for the physical drivers' interface, which in turn minimized the amount of memory needed for the kernel data.

Given the amount of memory allocated to the kernel space, the rest of the memory is available to the user processes, including our DGEMM driver code. Since we run our DGEMM driver at run-level one to avoid any unnecessary noise in the system, the only active processes at any one time were the DGEMM driver, the init process, the udev daemon and the shell. The *udev* daemon is a user daemon to serialize the events for device handling, and consumes 868 KB of the physical memory. *init* is the parent process of all user processes in the system, and consumes 648 KB of the physical memory. The shell consumes 1420 KB of the physical memory. Therefore, as the init process, udevd and shell consumes collectively 2.8 MB, the rest of the memory is available for our DGEMM driver code to use.

To monitor the memory consumption of the DGEMM driver in physical memory, we tracked its *R*esident *S*et *S*ize (*RSS*). The *RSS* reflects the number of pages the process has in real memory, minus 3 pages for administrative purposes. *RSS* only includes the pages that count towards text, data, or stack space of the process. This does not include pages that have not been demand-loaded in, or which are swapped out. The *RSS* growth of the DGEMM driver code is shown in Fig. 10. In this figure, the Y-axis represents the resident set size in KB as a function of the matrix size on the X-axis for different OS kernels. From this figure, we were able to characterize the memory pressure the DGEMM driver code is placing on the different OS kernels. The three 756 MB kernels have a growing *RSS* as the driver memory demand grows. On the other hand, the *RSS* growth of the 256 MB kernels ceases at matrix dimension 3100, both for the native kernel and the paravirtualized kernels. This effect is caused by the limited main memory allocated to the kernels, in which case the OS kernel would start swapping out por-

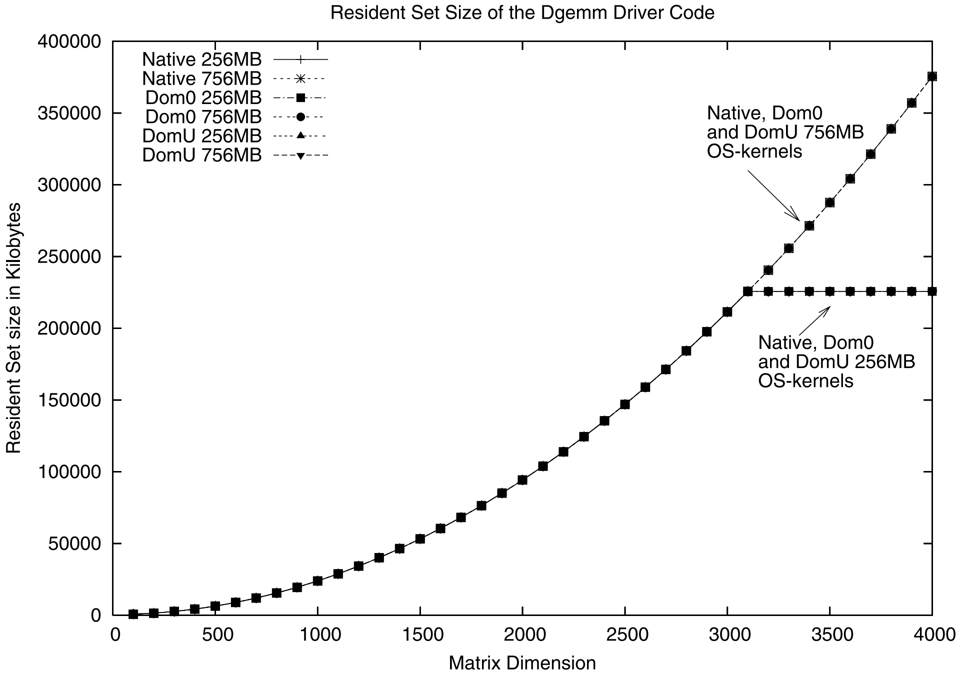


Fig. 10 The number of pages the DGEMM driver has in real memory

tions of the process address space in order to keep a minimal amount of memory space available for the other processes as well as the kernel.

5.3 Swap disk activity

In addition to the *RSS*, we investigated the swap disk activity in order to characterize its impact on the attainable performance. Towards this end, we monitored the swap disk activity and recorded the number of major and minor faults for each process in order to depict the MMU activity and the memory replacement policy between the native system and the paravirtualized system policies.

As Fig. 10 has shown, the amount of physical memory for the matrix data is limited for the OS kernels with 256 MB memory allocation, thus the OS kernel would start swapping out portion of the process address space. We monitored the swap disk activity using the Linux *free* command, which is portrayed by Fig. 11. It is important to note that the swap disk activity here is reflecting all kernel swap activity and not just the DGEMM driver code alone. For the three 756 MB OS kernels, there was no swap activity. This is expected since the 756 MB kernels had extended memory and have no need to swap the driver process out of memory. On the other hand, the 256 MB kernels have a different swapping activity, where they start swapping at matrix dimension of 3100 as illustrated in Fig. 11. Naturally, this is explained by the limited memory size allocated for those OS kernels. Therefore, Figs. 10 and 11 confirms that the performance drop in the DGEMM driver code for 256 MB kernels was caused by the swapping activity of the kernels.

In addition, we evaluated the MMU policy replacement for the kernels by studying the number of the major and minor faults as a function of the matrix dimension. Figure 12 depicts the number of major page faults (on the Y-axis) as a function of the matrix dimension (on the X-axis). Notice that each page is 4096 bytes. We observe that all the 256 MB kernels start swapping the DGEMM process address space at matrix dimension 3100.

Likewise, measuring the minor page faults can illustrate the behavior of the MMU and the replacement policy. A minor fault is a page that was marked for eviction by the replacement policy, but which was used by the user process before it was actually evicted. Figure 13 reflects the number of minor page faults in the system as a function of the matrix dimension. Again, there was no significant difference in the behavior of all the 756 MB kernels. There was no significant difference between the behavior of all the 256 MB kernels. However, we noticed a difference in the number of minor page faults between the 256 MB kernels and the 756 MB kernels. For the 256 MB kernels, the replacement policy is more aggressive in evicting pages from the main memory due to the memory pressure of the DGEMM process as the memory becomes more scarce in the system. Therefore, we noticed that the number of minor faults for the three 256 MB kernels increased tremendously as the DGEMM driver grew beyond matrix dimension of 3100, as a result of memory scarcity. Yet, there was no significant difference between the native and paravirtualized kernels in their MMU behavior, both in the scarcity and abundance of memory.

Fig. 11 The swap disk activity shown as the number of swapped bytes

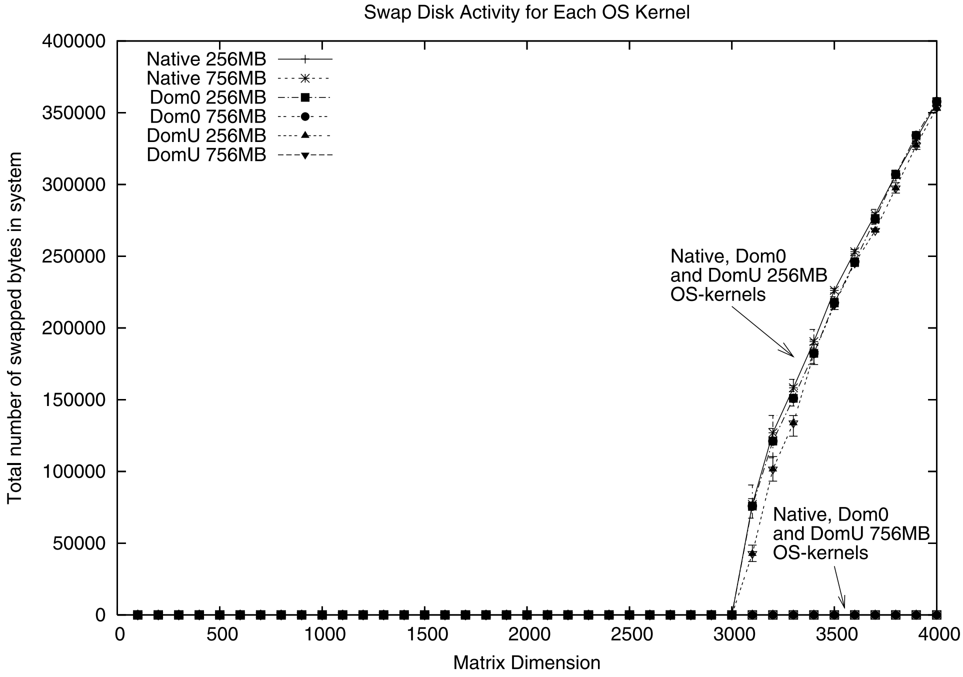
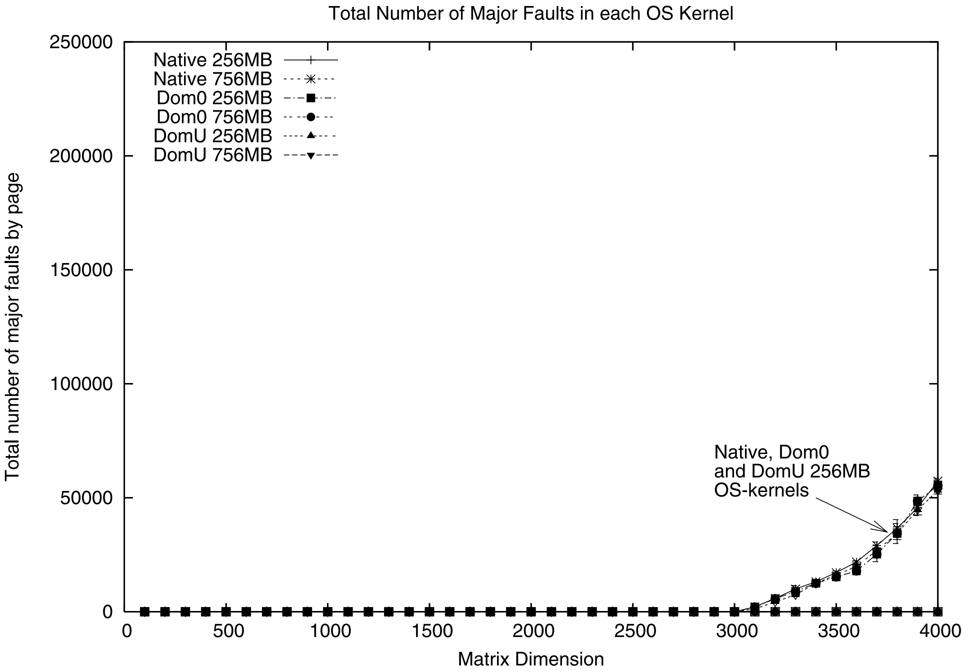


Fig. 12 The number of major faults caused by the DGEMM driver code



5.4 TLB activity and performance

The final level in the memory hierarchy that we investigated was the *T*ranslation *L*ookaside *B*uffer (*TLB*). We present the *TLB* performance for the various OS kernels under different memory configurations in this subsection. We used Oprofile [19], which is a low-overhead system-wide profiler that uses hardware counters to collect system statistics. Although there are several other similar profilers, Oprofile is the only profiler ported to Xen to profile the hypervisor, *Dom*0 and *DomU*. We measured two hardware events: the instruction *TLB* misses (ITLB), and the data *TLB* misses (DTLB). The data collected from the DGEMM driver is shown in Figs. 14 and 15.

The *TLB* size on the Pentium machines, which is the architecture we used in our experiments, is 64 entries for data *TLB* and 32 entries for instruction *TLB*. Given our memory

Fig. 13 The number of minor faults caused by the DGEMM driver code

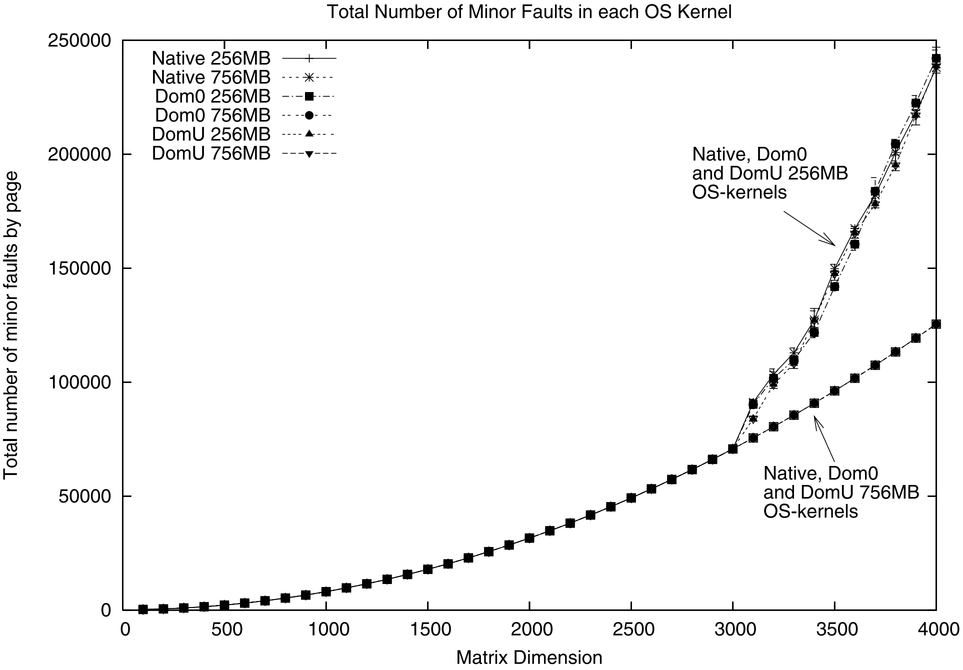
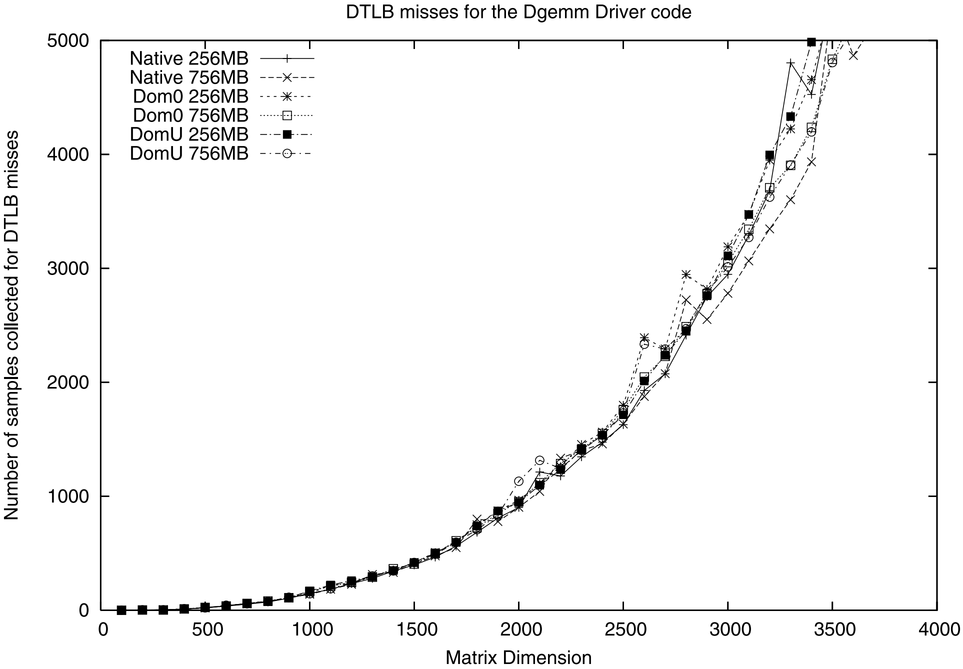
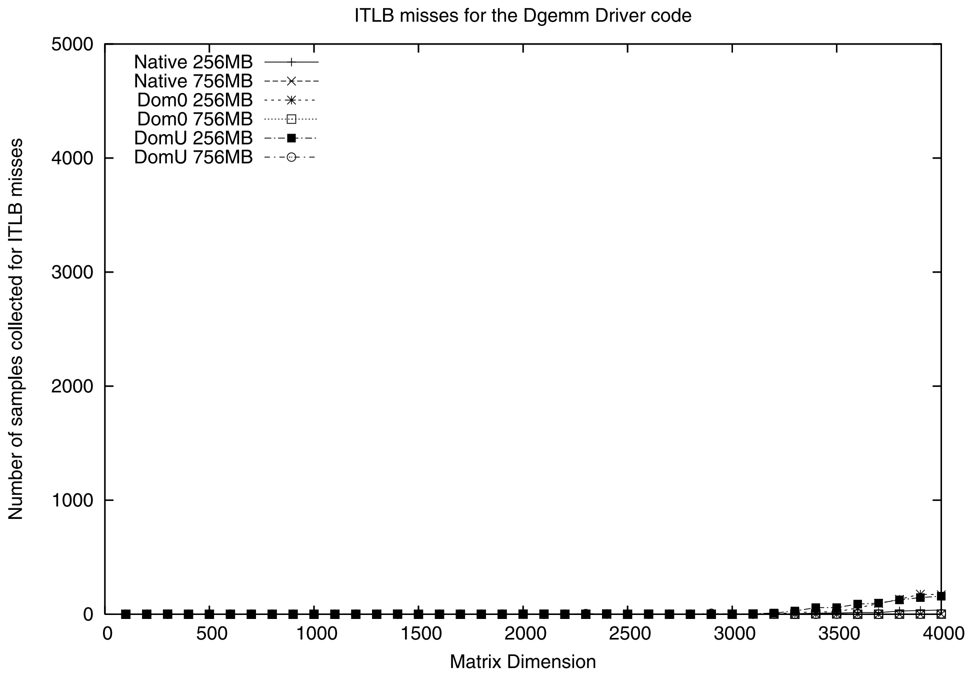


Fig. 14 The number of Data *TLB* misses as measured by Oprofile



intensive DGEMM driver, the rate of growth of the data is much faster than the size of the data *TLB*. This leads to having the number of DTLB misses to grow as the DGEMM process size grows, and the DTLB curve to follow a $3x^2$, the same function for data growth for the DGEMM driver data. Interestingly, There was no significant difference between the OS kernels based on their main memory allocation. In contrast with DTLB number of misses, the ITLB misses for DGEMM driver code is minimal as shown in Fig. 15, which is due to the small size of code section of the DGEMM driver. However, we noticed that ITLB misses started to increase at matrix dimension 3100 for the OS kernels with 256 MB. The ITLB misses start going up at the same time the swapping activity starts going up. We believe the extra swapping has resulted in more context switches between user code and system code and thus more instruction *TLB* entries expired, which in turn affected the measured ITLB activity for 256 MB OS kernels. However, there was

Fig. 15 The number of Instr *TLB* misses as measured by Oprofile



no difference in their *TLB* activity between the native and paravirtualized kernels with the same memory allocation.

6 Multiple threads performance

In the previous set of experiments, we studied the performance of a memory-intensive application and its influence on the paravirtualized kernel under memory pressure. We presented the performance difference between the native and paravirtualized kernels with similar main-memory allocation. Our results illustrate that paravirtualization has no significant impact on the performance of memory-intensive applications, even when memory becomes a scarce resource.

The next logical inquiry would question the performance ramification of paravirtualization on multiple concurrent DGEMM threads. In this respect, we are questioning the performance difference between running several threads in the same user space, and running each thread inside its own isolated virtual machine. This is an important investigation to tackle, since the latter model posses several benefits like encapsulation and fault-isolation. Furthermore, this model simplifies dynamic load balancing, as it enables OS-migration which allows virtual machines to be allocated to other physical nodes, transparently to the user process. For those benefits, we believe that executing several DGEMM threads in their separate VMs is an advantageous model. However, the interference between the different VMs running memory-intensive processes might degrade the performance harnessed by the DGEMM process. In addition, it is expected that concurrent processes in different VMs would encounter higher overhead, since the overhead of executing separate OS kernels in VMs is higher than the overhead of executing threads. For that, we designed another two experiments to investigate the potential impact of the interference on several concurrent DGEMM processes.

In order to evaluate the performance overhead of running concurrent DGEMM processes in separate VMs, we compare their performance with the same number of concurrent DGEMM processes running in the same VM. In order to ensure a fair comparison, we allocate the sum of main-memory of the separate VMs to the one VM running the DGEMM processes. To simplify our experimentation, we run our tests with two concurrent DGEMM processes. The results of our experimentation are shown in Fig. 16. The subfigure on the left portrays the average Mflops of two concurrent DGEMM processes executing in the same VM, which runs a SMP-enabled OS kernel with two CPUs, and 512 MB memory allocated at startup. On the other hand, the subfigure on the right portrays the average Mflops of two concurrent DGEMM processes running in two separate VMs with 256 MB main-memory each. For each of the two VMs, we ensured to allocate one physical CPU for each, in order to isolate the effect of context-switching between physical CPUs.

For both experiments, we observed that both performance curves increase until they exceed 4 Gflops, when the performance stabilizes for both curves equally. This essentially mirrors the single process results shown in Sect. 5 for the same amount of memory. Afterwards, we observe that the right subfigure performance curve degrades much slower with lower variance at matrix dimension 3100 than the left

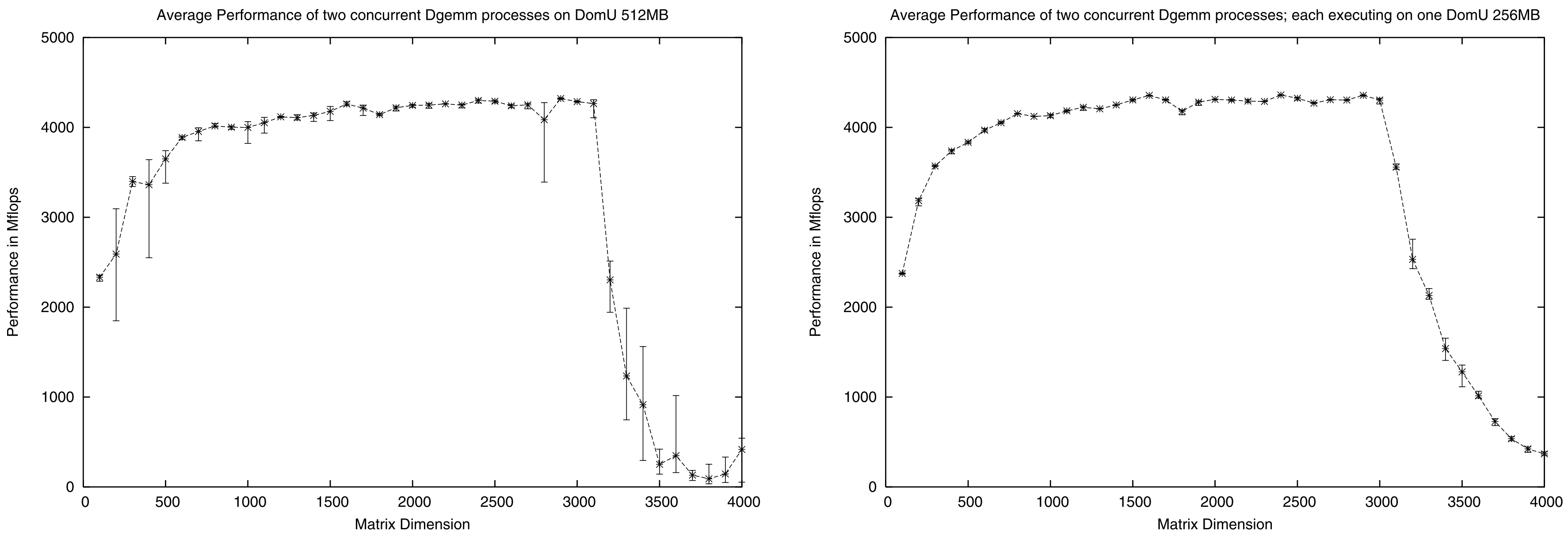


Fig. 16 The performance in Mflops of two concurrent DGEMM driver code threads as a function of the matrix dimensions. The *left subfigure* portrays the average performance of the two DGEMM processes for a kernel with 512 MB memory, while the *right subfigure* represents the performance of two DGEMM processes running in two VMs, each VM with 256 MB

subfigure. Furthermore, the total achieved Mflops by the two process in the same VM (i.e. left subfigure) are negatively impacted.

We believe that this is a result of the memory and performance isolation offered by virtualization. To explain, since the two processes are sharing the same user-space memory, they therefore are prone to the overhead of context switching. In addition, as the affinity of processes running in the same VM is not fixed to specific CPUs, the OS kernel is context-switching the processes to more than one CPU during its execution time. This, in turn randomly prolongs the overall execution time of the processes, which is reflected by the high variance in data, as shown in the left subfigure. On the other hand, we believe that the lower variance in the separate VMs (i.e. right subfigure) is caused by the fixed affinity of the CPUs of the VMs. To sum, these experiments show that concurrent memory-intensive processes in separate VMs can achieve at least similar, or even better performance to threads sharing the same VM, due to the performance isolation between VMs and ability to control CPU affinity of virtual machines. Although these experiments do not present a comprehensive examination of this performance difference, it illustrates the potential performance and functionality advantages of encapsulating concurrent memory-intensive processes in distinct VMs.

7 Inter VM communication

In this section, we will consider the performance impact of para-virtualization on applications that communicate across different virtual machine domains.

Current virtualization systems are specifically designed to isolate processes running in different operating systems from each other. With the possibility of over a hundred cores within a single HPC machine, however, both flexibility and scalability requirements make it necessary to be able to support—within the same machine—multiple operating systems that can be used simultaneously by a single large-scale application. Such applications will consist of "tasks" that exchange messages, but within which different forms of parallelism based on shared memory (e.g., MIMD or SIMD/vector) will be combined. To support such applications, we examined a memory-sharing mechanism for Xen-based virtual machines and evaluated its performance with micro-benchmarks.

The current approach to implementing memory isolation in Xen is to partition the available physical memory among hosted domains, guarding writes to page tables, but allowing standard virtual memory address resolution in the absence of page faults. When a guest OS needs to update its page tables, it must explicitly call Xen through a "hypercall" so that Xen can check to ensure the memory being updated belongs to that OS. To enable efficient communication between domains running on the same machine—e.g., when they establish a network connection—Xen offers "grant tables," which enable a guest OS to grant access to regions of their own memory to other guests. We evaluated the implementation of page sharing, a implemented with grant tables, so that individual memory pages can be shared among guest OSs.

We crafted three simple micro-benchmarks to measure the efficiency of memory sharing:

- *Raw*: passes control back-and-forth between two domains using a single shared binary variable. This benchmark measures the raw speed of memory sharing.
- *Sync*: uses classic P/V semaphores implemented using *bakery algorithm* [17] to enforce synchronized (and necessarily alternating) access to a shared integer variable. This benchmark demonstrates the performance of a more realistic fine-grained synchronization scenario.
- *TCP*: uses a socket with alternating `send()` and `recv()` invocations in each thread. Our intent was to measure the overhead of TCP connection management and Xen protection mechanisms.

We used our micro-benchmarks to measure the latency of communication between two threads as follows. For each benchmark run, the threads passed control back-and-forth for 10^6 times. We then computed the mean duration of an iteration (along with its standard deviation).

For each benchmark, we studied three cases. The first is memory sharing between processes executing under native Linux, the second is memory sharing between *Dom*0 and *DomU* OSs, and the third is memory sharing between two *DomU*s. In all cases we enable Linux processor affinity to minimize cache and *TLB* pollution effects. Finally, we were concerned about the possibility that enabling support for SMP threading in the host kernel might perturb the results. Thus we conducted the experiments both for the native host without SMP support and with it enabled. Table 6 summarizes the results. Each row of the table corresponds to a benchmark, with *Sync* appearing twice ($Sync_{1k}$ involved passing one thousand bytes from one thread to the other instead of a single integer).

From the table, we can see that memory sharing via modified grant tables under Xen (first three rows) proceeds at native speeds. It may appear that, in fact, Xen is faster (the first element in the first column is larger than the others). However, in this case, the memory sharing had to be between user-level processes (since the kernel is not multi-threaded). We included this test as a control of our measurement infrastructure as we would expect user-space to user-space transfers to be more expensive. In all other cases shown in the first three rows, however, the transfers are kernel-to-kernel and the data indicates that the speeds are the same.

Table 6 Latency of communication under *Raw*, *Sync*, and *TCP* micro-benchmarks. The units are microseconds, each number not in parentheses is the average over 10^6 iterations of 25 runs and the parenthesized number is the standard deviation of the 25 runs

	Native	host OS w/o SMP		host OS w/ SMP		
	Linux	host-guest	guest-guest	host-host	host-guest	guest-guest
Raw	0.49 (0.03)	0.36 (0.007)	0.36 (0.005)	0.36 (0.009)	0.35 (0.005)	0.39 (0.03)
Sync	1.4 (0.02)	0.8 (0.01)	0.8 (0.02)	0.8 (0.01)	0.78 (0.03)	0.8 (0.02)
$Sync_{1k}$	3.4 (0.07)	3.3 (0.05)	3.3 (1.14)	2.8 (0.15)	3.1 (0.2)	2.9 (0.1)
TCP	52.6 (0.4)	56.7 (0.2)	80.9 (0.3)	38.4 (0.3)	70.3 (0.4)	104.0 (0.5)

More rigorously, comparisons of the means using a t-test for all but the first value in the first row provides no evidence contradicting the assertion that the means are the same.

The values in the bottom two rows of the table allow us to speculate on how shared-memory communication compares to socket-based communication. While our $Sync_{1k}$ results are comparable to bulk-data bandwidth measurements reported in literature [32], the latency of fine-grained synchronization is lower when using shared memory, as our *TCP* experiment (in the last row of Table 6) illustrates. Message-passing in shared memory is at least 50 faster than in a socket.

These simple experiments indicate that it is possible to achieve memory sharing among OS instances at native execution speeds for multi-core systems using paravirtualization.

8 Discussion

In this paper, we have measured and analyzed the paravirtualization impact on performance in several ways. First, we examined its impact on autotuned *LA* routines as configured by *ATLAS*. Then, we examined the performance degradation when running multiple DGEMM instances under single and multiple VM scenarios. Finally, we looked at the performance of memory sharing between OS instances.

As it is of particular importance to highly-tuned *LA* code, we have focused on the different levels of the memory hierarchy and compared the memory characteristics between the native non-paravirtualized OS kernel and the paravirtualized kernels since paravirtualization affects the way in which virtual memory is manipulated. Our results show that there is no significant difference in performance between native execution and paravirtualized execution even when *ATLAS* tunes the performance of the libraries to near peak speeds. These results are quite remarkable since paravirtualization could easily impact virtual memory system activity (e.g. *TLB* miss rate) by introducing another level of process scheduling and I/O indirection. Further, one might expect these effects to manifest near the performance boundary of high-tuned numerical programs. However, as the major workload of the computational code uses non-privileged instructions that run at the native speed of the processor, it does not impact the overall tuning of the *ATLAS* routines. Although under heavy memory usage, the Xen hypervisor is frequently invoked to trap every page-table update, this overhead did not invest itself in the overall attainable performance by the DGEMM code.

Our results also demonstrated that the paravirtualization did not affect the processor characteristics detected by *ATLAS*. The kinds of characteristics that need to be detected will depend on the particular application being tuned. In the case of dense linear algebra, some of the important characteristics are cache size, number of registers, functional unit latencies, memory latency, etc. In some sense, the exact value of a certain characteristic is not as important as its effect on the performance of the generated code. For example, while it is useful to know that a processor has a cache size of 256 KB, the tuning system really wants to know how much data it can typically fit in a matrix block before incurring too many cache misses. Thus, the detected values can be viewed as good starting points or bounds for exploration of the search space, but not sufficient by themselves to generate the best code. Having the hardware detection coupled with an empirical parameter search allows the system to cope with any inaccuracies in the detection of the characteristics.

We have additionally shown that peak performance was not significantly degraded when running multiple instances of DGEMM under two virtualization scenarios: single VM with multiple threads and multiple VMs each with a single thread. This is an encouraging result since it would allow taking advantage of the potential benefits of isolating the tasks in separate VMs without incurring a serious performance penalty. As a practical illustration, data centers are already partitioning their resources into separate VMs which are assigned to different customers simultaneously.

In the previous scenario, the separate DGEMM instances had no need to communicate with each other, but for large-scale HPC applications, this will almost certainly be a necessity. We have addressed this issue by examining the performance of memory sharing between different OS instances, showing that native execution speeds are possible. This is a promising result for HPC application performance since most MPI implementations can use shared memory communications between processes on the same machine, but not necessarily across VMs. However, there is some recent research demonstrating good results with an MPI implementation enabled with inter-VM shared memory [14].

Therefore, our results show that the combination of *ATLAS* autotuning and Xen paravirtualization deliver native execution performance and nearly identical memory hierarchy performance profiles. Given that the host OS does not necessarily require a fully dedicated core, linear algebra software can take full advantage of the computational power of the physical hardware without being penalized for running in a paravirtualized environment. Furthermore, some previous results from our research and others confirmed that paravirtualization did not impact MPI communications performance over popular network infrastructure, like Ethernet and Infiniband [24]. This, in turn allows linear algebra routines to efficiently run in distributed virtual environments.

Our results expose a new potential benefit of paravirtualization for numerically and memory intensive applications. Having the performance of the DGEMM driver code dependent on the amount of memory available at user space, slimming and customization of the OS kernel to run as a virtual machine can possibly deliver better performance than native kernels, as a slimmed down kernel can be tuned to allow the user space process to have more memory than running on a native system (which is configured to run a full workload). In addition, our results support the feasibility of deploying linear algebra systems and memory intensive applications on virtualized systems. Consequently, this would support the practicality of utilizing virtual clusters [12] and cloud computing as a computing environment for those applications. For example, in Amazon's Elastic Cloud (EC2) [2] and IBM's Blue Cloud [15], the hosted kernels are specifically configured for numerical execution.

9 Conclusions

In this paper, we have examined the impact of paravirtualization on several aspects of performance relevant to HPC applications. We presented a comprehensive evaluation of the memory hierarchy characteristics of the paravirtualized kernels. We presented experimentation to show the impact of paravirtualization on empirically autotuned codes for linear algebra software. We also studied the performance of a memory-intensive application and its influence on the paravirtualized kernel under memory pressure. We presented the performance difference between the native and paravirtualized kernels with similar main-memory allocation. Then we investigated the performance degradation when running multiple instances of the same memory-intensive application under different VM scenarios. Finally, we examined the cost of inter-VM communication via shared memory.

Our results illustrate that paravirtualization has no significant impact on the performance of memory-intensive applications, even when memory becomes a scarce resource. Paravirtualization, furthermore, does not alter the system image and does not affect the ability of empirically tuned codes to produce peak performance for linear algebra software. Given the rise of the new paradigm of cloud computing, paravirtualization exposes new deployment scenarios for linear algebra computational kernels and software.

References

1. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann, Los Altos (2002)
2. Amazon: Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2 (2007)
3. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide. Society for Industrial and Applied Mathematics, Philadelphia (1999)
4. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: a view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec. (2006)
5. Back, G., Nikolopoulos, D.S.: Application-specific customization on many-core platforms: the VT-ASOS framework. In: Proceedings of the Second Workshop on Software and Tools for Multi-Core Systems, March 2007
6. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R.: Virtual machine monitors: Xen and the art of virtualization. In: Symposium on Operating Systems Principles (SOSP), 2003
7. Bilmes, J., Asanovic, K., Chin, C.-W., Demmel, J.: Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In: International Conference on Supercomputing, pp. 340–347, 1997
8. Blackford, L.S., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Kaufman, L., Lumsdaine, A., Petitet, A., Pozo, R., Remington, K., Whaley, R.C.: An updated set of Basic Linear Algebra Subprograms (BLAS). ACM Trans. Math. Softw. **28**(2), 135–151 (2002)
9. Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I., Warfield, A.: Live migration of virtual machines. In: USENIX Symposium on Networked Systems Design and Implementation (NSDI '05), Boston, MA, USA, May 2005
10. Demmel, J., Dongarra, J., Eijkhout, V., Fuentes, E., Petitet, A., Vuduc, R., Whaley, C., Yelick, K.: Self-adapting linear algebra algorithms and software. Proc. IEEE **93**(2), 293–312 (2005) (Special Issue on Program Generation, Optimization, and Adaptation)
11. Dongarra, J.J., Croz, J.D., Hammarling, S., Hanson, R.J.: An extended set of FORTRAN Basic Linear Algebra Subprograms. ACM Trans. Math. Softw. **14**(1), 1–17 (1988)
12. Foster, I., Freeman, T., Keahy, K., Scheftner, D., Sotomayer, B., Zhang, X.: Virtual clusters for grid communities. In: CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06). Washington, DC, USA, 2006, pp. 513–520. IEEE Computer Society, Los Alamitos (2006)
13. Frigo, M., Johnson, S.G.: FFTW: An adaptive software architecture for the FFT. In: Proc. 1998 IEEE Intl. Conf. Acoustics Speech

and Signal Processing, vol. 3, pp. 1381–1384. IEEE, New York (1998)

14. Huang, W., Koop, M., Panda, D.: Efficient one-copy MPI shared memory communication in virtual machines. In: IEEE Cluster 2008, 2008
15. IBM: IBM Blue Cloud. http://www-03.ibm.com/press/us/en/pressrelease/22613.wss, Nov. (2007)
16. Krintz, C., Wolski, R.: Using phase behavior in scientific application to guide Linux operating system customization. In: Workshop on Next Generation Software at IEEE International Parallel and Distributed Processing Symposium (IPDPS), April 2005
17. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Commun. ACM **17**(8), 453–455 (1974)
18. Lawson, C.L., Hanson, R.J., Kincaid, D.R., Krogh, F.T.: Basic Linear Algebra Subprograms for Fortran usage. ACM Trans. Math. Soft. **5**(3), 308–323 (1979)
19. Levon, J.: Oprofile—a system profiler for Linux. http://oprofile.sourceforge.net/ (2004)
20. Mergen, M.F., Uhlig, V., Krieger, O., Xenidis, J.: Virtualization for high-performance computing. SIGOPS Oper. Syst. Rev. **40**(2), 8–11 (2006)
21. Nagarajan, A.B., Mueller, F., Engelmann, C., Scott, S.L.: Proactive fault tolerance for HPC with Xen virtualization. In: ICS '07: Proceedings of the 21st Annual International Conference on Supercomputing. New York, NY, USA, 2007, pp. 23–32. ACM, New York (2007)
22. Naughton, T., Vallee, G., Scott, S.: Dynamic adaptation using Xen. In: First Workshop on System-level Virtualization for High Performance Computing (HPCVirt 2007), March 2007
23. Padua, D.A., Wolfe, M.: Advanced Compiler Optimizations for Supercomputers. Commun. ACM **29**(12), 1184–1201 (1986)
24. Ranadive, A., Kesavan, M., Gavrilovska, A., Schwan, K.: Performance implications of virtualizing multicore cluster machines. In: Workshop on HPC System Virtualization, in Conjunction with Eurosys'08, Glasgow, UK, 2008
25. Ruth, P., Rhee, J., Xu, D., Kennell, R., Goasguen, S.: Autonomic live adaptation of virtual computational environments in a multidomain infrastructure. In: Autonomic Computing, 2006. ICAC '06. IEEE International Conference, pp. 5–14, 2006
26. Vuduc, R., Demmel, J., Yelick, K.: OSKI: a library of automatically tuned sparse matrix kernels. In: Proc. SciDAC 2005, Journal of Physics: Conference Series, vol. 16, San Francisco, CA, June 2005
27. Whaley, R.C., Petitet, A., Dongarra, J.: Automated Empirical Optimizations of Software and the ATLAS Project. Parallel Comput. **27**(1–2), 3–35 (2001)
28. Whitaker, A., Shaw, M., Gribble, S.: Scale and performance in the Denali isolation kernel. In: Symposium on Operating Systems Design and Implementation (OSDI), 2002
29. Youseff, L., Wolski, R., Gorda, B., Krintz, C.: Evaluating the performance impact of Xen on MPI and process execution for HPC systems. In: VTDC '06: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing, 2006
30. Youseff, L., Wolski, R., Gorda, B., Krintz, C.: Paravirtualization for HPC systems. In: Min, G., Martino, B.D., Yang, L.T., Guo, M., Rünger, G. (eds.) ISPA Workshops. Lecture Notes in Computer Science, vol. 4331, pp. 474–486. Springer, Berlin (2006)
31. Youseff, L., Wolski, R., Krintz, C.: Linux kernel specialization for scientific application performance. Technical Report UCSB Technical Report 2005-29, Univ. of California, Santa Barbara, Nov. (2005)
32. Zhang, X., McIntosh, S., Rohatgi, P., Griffin, J.L.: Xensocket: a high-throughput interdomain transport for vms. Technical report, IBM Research Technical Report RC24247 (2007)

Lamia Youseff is currently a Ph.D. candidate in computer science at the University of California, Santa Barbara (UCSB). Her general research interests include operating systems, high performance computing and virtualization techniques. Her doctoral research is focused on virtualization aspects for scientific and HPC applications, under the supervision of Prof. Rich Wolski at UCSB. Prior to joining UCSB, Lamia received her B.Sc. from The American University in Cairo (AUC), with Summa Cum Laude. During her undergraduate tenure, she received several awards including the international ACM-UPE Scholarship Award in 2002 and AUC President's cup in 2003.

Keith Seymour is a Senior Research Associate at the Innovative Computing Laboratory (ICL) in the Electrical Engineering and Computer Science Department at the University of Tennessee, where he has worked on the f2j/JLAPACK, NetSolve/GridSolve, PAPI, and GCO (Generic Code Optimization) projects. He received his M.S. in Computer Science from the University of Tennessee in 1997. His research interests include grid computing and empirical code optimization.

Haihang You is a Research Associate at the Innovative Computing Laboratory (ICL) in the Computer Science Department at the University of Tennessee, Knoxville. He received his M.S. in Computer Science and M.S. Minor in Physics from the University of Tennessee in 2001, B.S. in Physics from Beijing Normal University. His research interests include Performance Analysis, Empirical Code Optimization, and Adaptive Finite Element Methods for Differential Equations.

Dmitrii Zagorodnov received the B.S. and M.S. degrees in computer science from the University of Alaska, Fairbanks, and the Ph.D. degree from the University of California San Diego. He is currently a researcher at the Computer Science Department of the University of California Santa Barbara. His research interests are the area of software systems: operating system design, network protocols and routing, as well as theoretical and applied problems in distributed systems.

Jack Dongarra received a Bachelor of Science in Mathematics from Chicago State University in 1972 and a Master of Science in Computer Science from the Illinois Institute of Technology in 1973. He received his Ph.D. in Applied Mathematics from the University of New Mexico in 1980. He worked at the Argonne National Laboratory until 1989, becoming a senior scientist. He now holds an appointment as University Distinguished Professor of Computer Science in the Computer Science Department at the University of Tennessee and holds the title of Distinguished Research Staff in the Computer Science and Mathematics Division at Oak Ridge National Laboratory (ORNL), Turing Fellow at Manchester University, and an Adjunct Professor in the Computer Science Department at Rice University. He is the director of the Innovative Computing Laboratory at the University of Tennessee. He is also the director of the Center for Information Technology Research at the University of Tennessee which coordinates and facilitates IT research efforts at the University.

He specializes in numerical algorithms in linear algebra, parallel computing, the use of advanced-computer architectures, programming methodology, and tools for parallel computers. His research includes the development, testing and documentation of high quality mathematical software. He has contributed to the design and implementation of the following open source software packages and systems: EISPACK, LINPACK, the BLAS, LAPACK, ScaLAPACK, Netlib, PVM, MPI, NetSolve, Top500, ATLAS, and PAPI. He has published approximately 200 articles, papers, reports and technical memoranda and he is coauthor of several books. He was awarded the IEEE Sid Fernbach Award in 2004 for his contributions in the application of high performance computers using innovative approaches and in 2008 he was the recipient of the first IEEE Medal of Excellence in Scalable Computing. He is a Fellow of the AAAS, ACM, and the IEEE and a member of the National Academy of Engineering.

Rich Wolski is a Professor in Computer Science at the University of California, Santa Barbara (UCSB). He received his M.S. and Ph.D. degrees from the University of California at Davis (while he held a full-time research position at Lawrence Livermore National Laboratory). He is currently also a strategic advisor to the San Diego Supercomputer Center and an adjunct faculty member at the Lawrence Berkeley National Laboratory. His most recent research efforts have focused on the development of statistical predictive techniques for resource-constrained power usage, resource failure prediction, batch queue delay prediction, and cloud computing infrastructure design and implementation.