MuMMI: Multiple Metrics Modeling Infrastructure

Xingfu Wu
Charles Lively
Valerie Taylor
Dept. of Computer Science & Engineering
Texas A&M University
College Station, TX 77843
Email: {wuxf, clively, taylor}
@cse.tamu.edu

Hung-Ching Chang
Chun-Yi Su
Kirk Cameron
Dept. of Computer Science
Virginia Tech
Blacksburg, VA 24061
Email: {hcchang, sonicat, cameron}
@vt.edu

Shirley Moore
Dan Terpstra
and Vince Weaver
Department of EECS
University of Tennessee
Knoxville, TN 37996
Email: {shirley, terpstra, vweaver1}
@eecs.utk.edu

***Abstract*—The MuMMI (Multiple Metrics Modeling Infrastructure) project is an infrastructure that facilitates systematic measurement, modeling, and prediction of performance, power consumption and performance-power tradeoffs for parallel systems. In this paper, we present the MuMMI framework, which consists of an Instrumentor, Databases and Analyzer. The MuMMI instrumentor provides for automatic performance and power data collection and storage with low overhead. The MuMMI Databases store performance, power and energy consumption and hardware performance counters' data. The MuMMI Analyzer entails performance and power modeling and performance-power tradeoff and optimizations. As part of the MuMMI project, we mainly focus on discussing the design and development of a MuMMI Instrumentor to provide automatic performance and power data collection and storage with low overhead on multicore systems in detail, then utilize the MuMMI Instrumentor to collect performance and power data for a hybrid MPI/OpenMP earthquake application to discuss application performance-power trade-off and optimizations. Our experimental results show that we reduce up to 8.5% the application execution time and lower up to 18.35% the energy consumption by applying Dynamic Voltage and Frequency Scaling (DVFS), Dynamic Concurrency Throttling (DCT) and loop optimizations.**

Keywords: Performance measurement, Power measurement, Performance-Power optimization, Performance-power tradeoff, Dynamic Voltage and Frequency Scaling (DVFS), Dynamic Concurrency Throttling (DCT)

I. Introduction

The burgeoning revolution in high-end computer architecture has far reaching implications for the software infrastructure of tools for performance measurement, modeling, and optimization, which has been indispensable to improved productivity in computational science over the past decade. Significant work has been done on exploring power reduction strategies for large-scale, parallel scientific application [2], [3], [4], [6]. The problem is that much of this work required manual data collection and analysis to explore the different power reduction techniques. Very little work has been done with respect to providing an infrastructure for automating as much of this process as possible in addition to archival of the data. The MuMMI (Multiple Metrics Modeling Infrastructure) project [10] was developed to provide an infrastructure that facilitates systematic measurement, modeling, and prediction of performance, power consumption and performance-power tradeoffs for parallel systems.

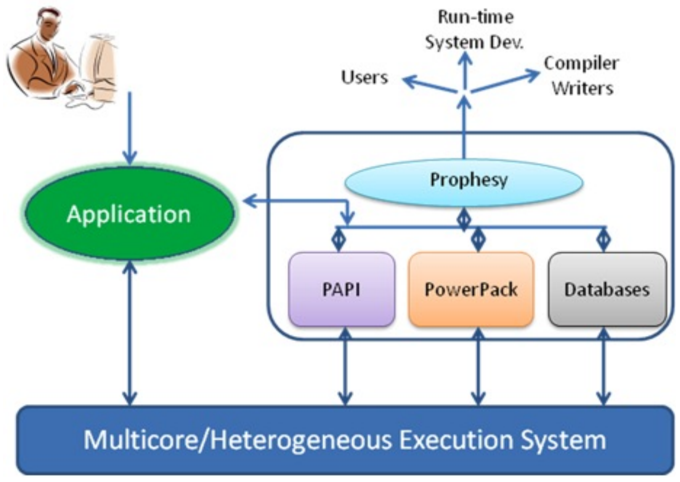


Fig. 1. Multiple Metrics Modeling Infrastructure (MuMMI).

MuMMI is depicted in Figure 1, as a multi-level infrastructure for integrated performance and power modeling for large-scale parallel systems. MuMMI builds upon an established infrastructure for performance modeling and prediction (Prophesy [16]), hardware performance counter measurement (PAPI [11]), and power profiling (PowerPack [4]). MuMMI extends these existing infrastructures in the following ways:

- Extension of Prophesy's well-established performance modeling interface to encompass multicores and GPUs and to incorporate power parameters and metrics into the performance models. A database component enables the modeling system to record and track relevant benchmark codes and results for characterizing multicore architectures.
- Building on top of PAPI's multi-component architecture, MuMMI provides a user-configurable layer for defining derived higher level metrics relevant to the performance modeling task at hand and mapping these to available native events on a given platform.
- Extension of an emerging power-performance measurement, profiling, analysis and optimization framework, called PowerPack, to multicore architectures. This work enables MuMMI to measure and predict power consumption at component (e.g. CPU, memory

and disk) and function-level granularity for parallel architectures.

As a result of these combined efforts, MuMMI is able to offer an integrated and extensible performance modeling and prediction framework for use by application developers, system designers, and scientific application users, helping them to make key choices for configuring and selecting appropriate parallel systems and to evaluate and optimize power/performance on these systems.

In this paper, we present the MuMMI framework which consists of three main components: Instrumentor, Databases and Analyzer. The MuMMI instrumentor provides for automatic performance and power data collection and storage with low overhead. The instrumentor has been used for our recent research work in [7], [8], [9]. MuMMI Databases extend the databases of Prophesy to store power and energy consumption and hardware performance counters' data. The MuMMI Analyzer extends the data analysis component of Prophesy to support power consumption and hardware performance counters, and it entails performance and power modeling and performance-power tradeoff and optimizations. In this paper, as part of MuMMI project, we mainly focus on discussing the design and development of a MuMMI Instrumentor to provide automatic performance and power data collection and storage with low overhead on multicore systems in detail, then utilize the MuMMI Instrumentor to collect performance and power data for a hybrid MPI/OpenMP earthquake application to discuss application performance-power trade-off and optimizations. Our experimental results show that we reduce up to 8.5% the application execution time and lower up to 18.35% the energy consumption by applying Dynamic Voltage and Frequency Scaling (DVFS), Dynamic Concurrency Throttling (DCT) and loop optimizations.

The remainder of this paper is organized as follows. Section 2 describes the MuMMI framework. Section 3 presents the design framework and implementation of the MuMMI Instrumentor in detail, and discusses the overhead and limitation of the instrumentor. Section 4 uses a parallel earthquake simulation as an example to discuss the MuMMI Analyzer, especially focusing on performance-power trade-off and optimization based on the performance and power data collected by the MuMMI Instrumentor. Section 5 summarizes the paper.

II. MuMMI Framework

In this section, we discuss the four main components of the MuMMI framework shown in Figure 1: PAPI, PowerPack, Prophesy, and MuMMI database.

A. PAPI

Hardware performance monitors can provide a window on the hardware by exposing detailed information on the behavior of functional units within the processor, as well as other components of the system such as memory controllers and network interfaces. The PAPI project designed and implemented a portable interface to the hardware performance counters available on most modern microprocessors [11], [1]. The PAPI specification includes a common set of PAPI standard events considered most relevant to application performance evaluation – cycle and operation counts, cache and memory access events, cache coherence events, and branch prediction behavior – as well as a portable set of routines for accessing the counters. Moreover, performance counter hardware has spread beyond just the CPU. Performance monitors can now be found in memory controllers, network fabrics, and thermal monitors, as well as on special purpose accelerators such as GPUs. The work to restructure the PAPI library contains the exposed API and platform independent code, and a collection of independently loadable components, with one component for each set of hardware monitoring resources. As part of MuMMI, this work allows the simultaneous monitoring and correlation of performance data from multiple components and levels of the system.

B. PowerPack

The PowerPack [4] is a collection of software components, including libraries and APIs, which enable system component-level power profiling correlated to application functions. PowerPack obtains measurements from power meters attached to the hardware of a system. The framework includes APIs and control daemons that use DVFS (dynamic voltage and frequency scaling) to enable energy reduction with very little impact on the performance of the system. As multicore systems evolve, the framework can be used to indicate the application parameters and the system components that affect the power consumption on the multicore unit. PowerPack allows the user to obtain direct measurements of the major system components' power consumption, including the CPU, memory, hard disk, and motherboard. This fine-grain measurement allows power consumption to be measured on a per-component basis.

C. Prophesy

The major component unifying the different components of MuMMI is the performance modeling and analysis component, which leverages from the Prophesy infrastructure [16]. Prophesy consists of three major components: data collection, data analysis, and three central databases [16], [17], [21], [22]. The data collection component focuses on automated instrumentation and application code analysis at the level of basic blocks, procedures, and functions. The resulting performance data are automatically stored in the performance database. Manual data entry is also supported. The data analysis component [22] produces analytical performance models with coefficients for a set of dependent variables at the granularity specified by the user (e.g., entire code, specific routines or code regions). The models are developed based upon performance data from the performance database, model templates from the template database, and system characteristics from the systems database. With respect to the performance models, in Prophesy we have used two well known techniques, curve fitting and parameterization models, and our technique developed, kernel coupling [15], [20]. The research work for MuMMI focuses on extending the performance modeling and database components to multicore and GPU systems and on incorporating power analysis into the models.

D. MuMMI Database

The MuMMI database facilitates contributions from and sharing among the high performance computing research community. In particular, MuMMI leverages from the Prophesy

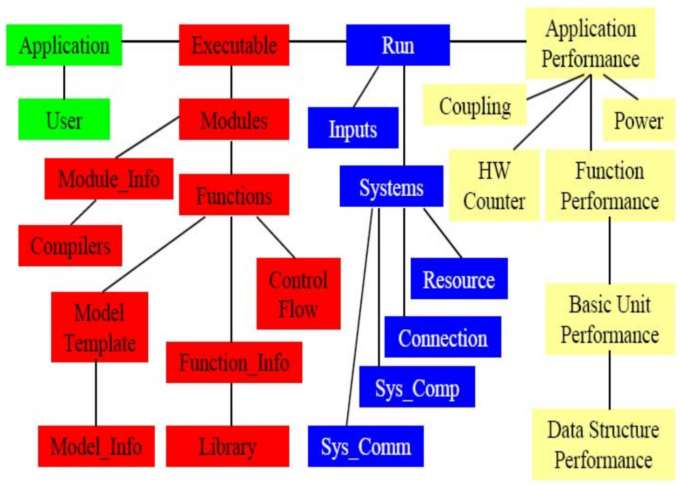


Fig. 2. MuMMI Database Schema.

database [21], [17] that allows for web-based entry of data in addition to automatic upload of data via SOAP.

The Prophesy database is composed of three major components: a performance database for detailed application performance data, a system database for detailed system information and performance data for low-level operations (such as floating point operations or communication primitives), and a template database for storing hints useful for generating models (e.g., the computational complexity of different operations).Together, the three Prophesy databases are organized into four areas: application information, executable information, run information, and performance statistics. The application information includes details about the application (e.g., version number, description, etc.) and application developer. The executable information includes all of the entities related to generating an executable of an application, including compiler and compiler flags. The run information includes all of the entities associated with running an executable and the system used for execution. Lastly, the performance statistics includes the performance data generated during execution of the application.

MuMMI database is the extension of the Prophesy database, which includes additional information relevant to power profiling, energy and hardware counters' performance on multicore platforms. The MuMMI database schema is shown in Figure 2. As we discussed in previous section, at the end of the instrumented program execution, performance and power data files are generated. These data files include the information stored in the MuMMI database. The information about user and application includes User name, password, email, application name and application version. The performance data for each executable includes executable name, problem size, total number of processors, total execution time, start timestamp, end timestamp, processor number, system name and run date. The performance data for each event (procedure or loop) includes Event ID, start line number, end line number, event type (Procedure or Loop), caller name, module name, runtime, number of calls, and so on.

The hardware counters' data from PAPI includes Cores per socket, Total Cores per node, Number of sockets per node, Core speed, Vendor and Model, PAPI_TOT_INS, PAPI _TOT _CYC, PAPI_L1_TCA, PAPI_L2_TCA, PAPI_L1_TCM, PAPI _L2_TCM, PAPI_TLB_TL, PAPI_BR_INS, PAPI_FP_INS, PAPI _FP_OPS, PAPI_BR_MSP, PAPI_L1_DCM, PAPI_L3 _TCM, PAPI_STL_ICY, PAPI_RES_STL, and so on. PAPI Multiplexing [11] allows a user to count more events than total physical counters recorded simultaneously by a system (Note that, most system architectures only allow to read four counter events at a time using cputrack.)

The power and energy data from PowerPack 3 includes system energy, CPU energy, memory energy, masterboard energy, disk energy, and power over time for each component such as system power over time, CPU power over time, memory power over time, masterboard power over time, disk power over time, and fan power over time. The power and energy data are for a whole application. Currently PowerPack is being extended to support power profiling at function level of the application. When the functionality is ready, the MuMMI database can be easily extended to store function-level power data.

III. MuMMI Instrumentor

Generally, collecting consistent performance and power data is difficult because it requires multiple executions of an application using each PAPI, PowerPack and Prophesy. Further, in some cases the instrumentation is done manually. For example, with PAPI, the subroutines are manually inserted to a source code in order to collect some performance counts events. With PowerPack, there is a need to manually start and stop the tool for power data collection. Our goal with the MuMMI Instrumentor is to make performance and power data collection easy and automatic requiring only one application execution for the collection of performance and power data. To the best of our knowledge, this is the first tool to support automatic and simultaenous performance, power and energy data collection.

In this section, we present the MuMMI Instrumentor framework and discuss its design and implementation in detail. The MuMMI Instrumentor provides a way to do source-code level automatic instrumentation for Fortran77, Fortran90, C and C++ programs. It is an extension of Prophesy data collection component [17] with the addition of power and hardware counters data collection and dynamic CPU frequency scaling support. The basic organization of the MuMMI Instrumentor is illustrated in Figure 3, which entails automatically inserting instrumentation code into a source code file. Support is provided for the following instrumentation options:

-a:	Instrument all procedures and outer loops
-p:	Instrument all procedures
-l:	Instrument all loops
-n:	Instrument all procedures not nested in loops
-F:	Use Perl SOAP scripts to automatically transfer performance data to the MuMMI database
-W:	Use PowerPack to collect power data
Default:	Instrument procedures and outer loops

These instrumentation options (-a, -p, -l, -n, or default) can be used to instrument a source code at different levels based on the user, ranging from only procedures and outler loops to nested loops. The default instrumentation option is

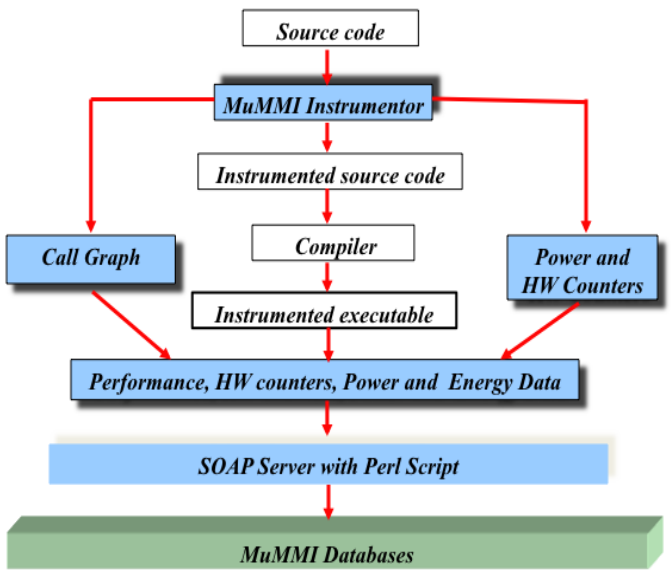


Fig. 3. MuMMI Instrumentor Framework.

to instrument all procedures and all loops for all source files. The user also can select the source files to be instrumented. However, the following routines must be instrumented: the main file such as *main()* function for a C program or the *PROGRAM* for a Fortran program. The MuMMI Instrumentor counts each procedure and loop in the source code, and labels it with a unique instrumentation point number (or called event identifier).

Recall that the MuMMI Instrumentor works with the source code to generate the corresponding instrumented file. The instrumented routines are renamed. For example, if the source code contains the main routine, it renames the C main function to *_MuMMI_main_()*. It also generates an extra initial main file *XX_init.c* containing a new *main()*. In the instrumented Fortran code, the original main routine like *PROGRAM NAME* is substituted with *SUBROUTINE NAME* and *XX_init.c* contains a call to *NAME_ or NAME* (based on different C/Fortran call interfaces). Note that, in Fortran90, the module/array has the same syntax as function calls like *YY(...)*. If instrumenting the function calls is allowed, the MuMMI Instrumentor will automatically instrument all the function calls plus the data structures with form of *YY(...)*. If the program is large, and has a lot of modules, there will be a lot of code unnecessarily instrumented. Therefore, for Fotran77 and Fortran90 programs, the MuMMI Instrumentor only instrument subroutines. In the instrumented C code, the original main function *main()* is replaced with *_MuMMI_main_()* and *XX_init.c* contains a call *_MuMMI_main_()*.

For the case when the "-W" option is used, the MuMMI Instrumentor records the starting and end timestamps of the program, and at the end of a program execution, it sends these timestamps to PowerPack 3.0 server to retrieve the power profiling data for the program, and appends the power and energy data to the output performance data files.

For the case when the "-F" option is used, at the end of the program execution, all performance data files are automatically transferred to the MuMMI database by the MuMMI Instrumentor. The MuMMI Instrumentor uses Perl SOAP script updatedb.pl we developed as a SOAP client side script to upload these data files to the MuMMI database (locally or remotely). We use SOAP Lite package (http://soaplite.com) to develop these scripts. Prior to invoking the SOAP client side script updatedb.pl to transfer the data files, the data files are converted to SQL scripts based on MuMMI database schema shown in Figure 2 so that the SOAP server side script dbserver.pl receives the data files, and uploads the data to the MuMMI database. Further, the MuMMI Instrumentor provides automatic upload of the information obtained from widely used performance analysis tools such as gprof, IPM [5], TAU [14] or Score-P [12] to the MuMMI database via some Perl SOAP scripts as well.

TABLE I. PAPI OVERHEAD (CYCLES) ON VARIOUS KERNEL INTERFACES

Linux Kernel	Start/Stop	Read	Read TS	Accum	Reset
2.6.32-pe	17749	8982	9019	11519	2550
2.6.35-pe	8683	2089	2109	4196	2084
2.6.30-perfmon	9644	1200	1202	3623	2406
2.6.32-perfctr	5702	195	203	3772	3556
SystemG-2.6.32-pe	16480	8387	8409	11425	2973

When the instrumented source code is generated, it requires the following compiler options to support different functionalities:

-DMPI:	Support MPI programs
-DPAPI:	Allow collecting hardware counters' data using PAPI
-DFS:	Support dynamic frequency scaling

It also requires PAPI library link -lpapi to support collecting hardware counters' data using PAPI. If the "-DFS" option is used, the MuMMI Instrumentor can support dynamic frequency scaling by adjusting the CPU frequency to what a user sets in the system file scaling_setspeed. It requires the user to specify CPU frequencies for scaling up or down in advance.

A. *Overhead and Limitations of MuMMI Instrumentor*

Because the MuMMI Instrumentor is a source code level instrumentation tool, it requires an application source code, and does not support dynamic instrumentation. However, after the instrumentor transforms the source code into the instrumented source code once, the instrumented code can be used on any platform.

As discussed previously, at the end of the program execution, the MuMMI Instrumentor uses Perl SOAP scripts to automatically process the data files obtained from Prophesy, PAPI and PowerPack, and puts the data into the MuMMI database. This design of the MuMMI Instrumentor does not affect the application performance. It only adds some post-processing overhead.

SystemG with 325 Mac Pro computer nodes from Virginia Tech is our testbed in this paper. Each node of the system contains two quad-core 2.8 GHz Intel Xeon processors and 8 GB memory. The CPU frequency for the Intel Xeon processor can only be adjusted to 2.4 GHz and 2.8 GHz. The linux kernel 2.6.32 and PAPI 4.1.2.1 are installed on SystemG.

In the following, we discuss the overhead costed by source-code level instrumentation, PowerPack and PAPI of the MuMMI Instrumentor:

1) Overhead from source-code level instrumentation
The MuMMI Instrumentor is a source-code level instrumentation tool. As we discussed in [17], some inline timing instrumentation codes are inserted to a source code based on a user's option in order to collect the performance and power data at the user specified level with less than 3.4% overhead based upon the NAS parallel benchmarks.

2) Overhead from PowerPack
For the power measurements, the PowerPack server runs on a separate machine, and the power measurement for PowerPack is external to the system used for execution. Therefore PowerPack does not add any overhead to the application performance.
For the case of the "-DFS" option, the MuMMI Instrumentor supports dynamic frequency scaling by adjusting the CPU frequency to what a user sets in the system file scaling_setspeed. It requires the user to specify CPU frequencies for scaling up or down in advance. For the overhead of CPU frequency scaling, we have to measure the overhead locally on each node because the current system only supports CPU frequency scaling for all cores on the same node, and does not support individual core frequency scaling. It means that if the frequency for one core is adjusted, the other cores on the same node must be adjusted to the same frequency.
The measured overhead for the scripts to adjust CPU frequencies for 8 cores on a node is 0.009 seconds. Currently, these shell scripts are command line. We plan to further reduce the overhead by using "write()" function to directly modify the system file scaling_setspeed to change the CPU frequency for each core.

3) Overhead from PAPI
PAPI is hardware performance counters' monitoring. There is some overhead from PAPI depending on various kernel interfaces. In [23], PAPI overhead on various kernel interfaces given in Table I is discussed by running the "papi_cost" utility program with PAPI 4.1.2.1 on a Core2 machine running four different kernels. The unit of reported values is average cycles. Where 2.6.32-pe stands for PAPI 4.1.2.1 with the linux kernel 2.6.32 and perf_events; 2.6.35-pe stands for PAPI 4.1.2.1 with the linux kernel 2.6.35 and perf_events; 2.6.30-perfmon stands for PAPI 4.1.2.1 with the linux kernel 2.6.30 and perfmon; 2.6.32-perfctr stands for PAPI 4.1.2.1 with the linux kernel 2.6.32 and perfctr; SystemG-2.6.32-pe stands for PAPI 4.1.2.1 with the linux kernel 2.6.32 and perf_events on our testbed SystemG.
As shown in Table I, the smaller the value is, the lower the PAPI overhead is. Note that, the overhead for 2.6.32-perfctr is the lowest except Reset; PAPI with the linux kernel 2.6.32 and perf_events has the highest overhead due to various workarounds that need to be done to properly set up the counters and read out the data. These poor behaviors were fixed by the linux kernel 2.6.35 so that the overhead for 2.6.35-pe is much lower than that for 2.6.32-pe. It is interesting to notice that the overhead for our SystemG-2.6.32-pe is lower than that for 2.6.32-pe except Reset. To further reduce the PAPI overhead, we plan to upgrade the linux kernel on SystemG to the latest linux kernel version 3.5.3, and install the latest version of PAPI (5.0). This is one simple way to reduce the PAPI overhead based on what we learned from Table I.

IV. MuMMI Analyzer

MuMMI Analyzer extends the data analysis component of Prophesy to support power consumption and hardware performance counters, and it entails performance and power modeling and performance-power tradeoff and optimizations. Our performance counters-based modeling methods [7], [9] identify the different performance counters that are needed to accurately predict the application performance, and provides insight to improve performance for the application. In this paper, we mainly focus on application performance-power trade-off and optimizations.

In this section, we use a parallel earthquake simulation application as an example. The example illustrates the collection and storage of the performance and power data into the MuMMI databases via the MuMMI Instrumentor, and the use of the performance and power data to analyze the trade-offs and optimizations among performance, power and energy consumption.

A. A MuMMI Testbed and A Parallel Earthquake Simulation

In this section, we describe a MuMMI testbed system, and deploy the MuMMI Instrumentor on the system. Our experiments utilize one multicore system from Virginia Tech, SystemG, which has 325 Mac Pro computer nodes. This system is the largest PowerPack-enabled research system with each node containing more than 30 thermal sensors and more than 30 power sensors. Each node has two quad-core 2.8 GHz Intel Xeon processors and 8 GB memory. The CPU frequency for the Intel Xeon processor can only be adjusted to 2.4 GHz and 2.8 GHz.

SCEC/USGS benchmark problem TPV210 is the convergence test of the benchmark problem TPV10 [13]. In TPV10, a normal fault dipping at 60 (30 km long along strike and 15 km wide along dip) is embedded in a homogeneous half space. Pre-stresses are depth dependent and frictional properties are set to result in a sub-shear rupture. This benchmark problem is motivated by ground motion prediction at Yucca Mountain, Nevada, which is a potential high-level radio-active waste storage site. In the benchmark problem TPV210, we conduct the convergence test of the solution by simulating the same problem at a set of element sizes, i.e., 200 m, 100 m, 50 m, and so on, where m stands for meters. Table II summarizes the parameters for TPV210. The benchmark requires larger memory sizes with finer element sizes, because the decrease of element size means the increase of numbers of elements and nodes. For example, for the element size of 50 m, it requires the number of elements be approximately 100,000,000, and the memory requirement is around 94 GB.

TABLE II. Parameters for TPV210

Parameters	Values		
Element size	200m	100m	50m
Total elements	6,116,160	24,651,088	98,985,744
Time step (s)	0.016	0.008	0.004
Termination(s)	15	15	15
Memory (GB)	∼6	∼24	∼94

The simulation is memory-bound. Our hybrid MPI/OpenMP finite element earthquake simulations discussed in [18], [19] target the limitation to reduce large memory requirements. For the sake of simplicity, we only use TPV210 with 200m element size as an example in this paper.

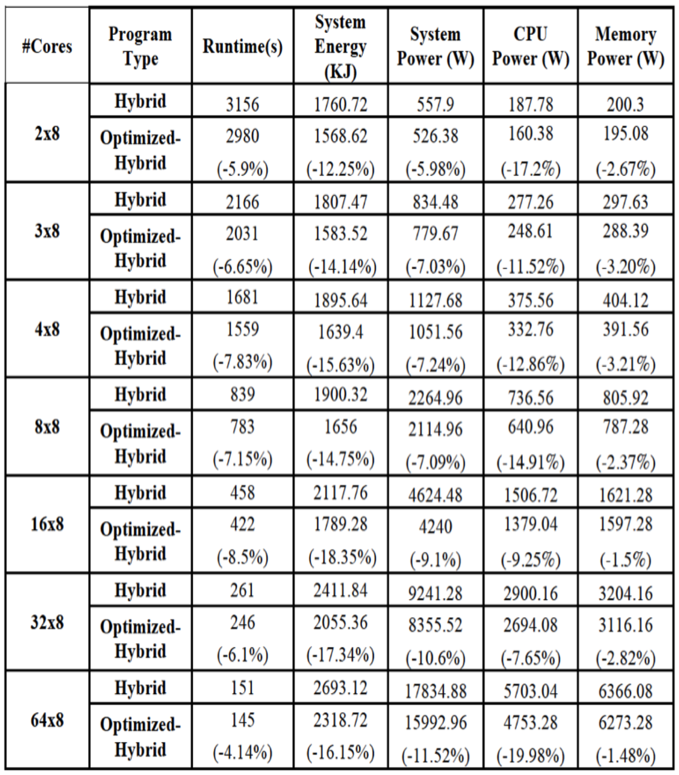
B. *Performance-Power Trade-off and Optimization*

In this section, we incorporate two common software-based approaches for reducing power consumption in large-scale parallel systems, Dynamic Voltage and Frequency Scaling (DVFS)[6] and Dynamic Concurrency Throttling (DCT) [2]. DVFS can be used to scale down the frequency of a HPC applications workload resulting in lower power consumption. DVFS is most beneficial when applied to regions within an application where communication does not overlap with computation. When DVFS is applied to applications that exhibit slack (lack of overlap between communication and computation) it results in reduced power consumption with minimal increases in application performance.

Dynamic concurrency throttling (DCT) is used to control the number of threads assigned to a segment of a parallel code. DCT can be applied to a code region with a reduced workload that would not benefit from using the maximum number of cores on a chip. Depending on the application and its workload requirements, utilizing fewer threads can reduce power consumption without impacting performance significantly. Applying DCT effectively results in reduced energy consumption with minimal increases in application performance for thread-based applications, such as OpenMP or hybrid applications.

For the hybrid MPI/OpenMP earthquake simulation, there is no overlapping between MPI communications and OpenMP parallel regions. The hybrid application consists of eight functions in its code: Input, qdct2, updated, qdct3, hourglass, communication, faulting and final, where qdct3 and hourglass are dominated. As we found in [19], the hybrid application is memory bound, and using 12 OpenMP threads per MPI process on Cray XT5 (with 12 cores per node) at Oak Ridge National Laboratory has more OpenMP overhead than using 4 OpenMP threads per MPI process on Cray XT4 (with 4 cores per node) at Oak Ridge National Laboratory for the hybrid execution on the same number of nodes with 1 MPI process per node although Cray XT5 is much faster than Cray XT4. In this paper, we test the hybrid application on SystemG, which has 8 cores per node and two frequency settings: 2.4GHz and 2.8GHz.

Before applying DCT and DVFS to the application executed on SystemG, we investigated how the settings of having 1, 2, 4, 6, or 8 OpenMP threads per node impact the application performance by utilizing the MuMMI Instrumentor to collect the performance and power consumption data, and found using 2 OpenMP threads per node for the functions qdct3 and hourglass results in the best application performance. So DCT is applied to the functions hourglass and qdct3 so that they are executed using 2 OpenMP threads per node during the application execution. DVFS is applied to the functions input, communication, and final for the communication slacks by adjusting CPU frequency from 2.8GHz to 2.4GHz.



#Cores	Program Type	Runtime(s)	System Energy (KJ)	System Power (W)	CPU Power (W)	Memory Power (W)
2x8	Hybrid	3156	1760.72	557.9	187.78	200.3
	Optimized-Hybrid	2980 (-5.9%)	1568.62 (-12.25%)	526.38 (-5.98%)	160.38 (-17.2%)	195.08 (-2.67%)
3x8	Hybrid	2166	1807.47	834.48	277.26	297.63
	Optimized-Hybrid	2031 (-6.65%)	1583.52 (-14.14%)	779.67 (-7.03%)	248.61 (-11.52%)	288.39 (-3.20%)
4x8	Hybrid	1681	1895.64	1127.68	375.56	404.12
	Optimized-Hybrid	1559 (-7.83%)	1639.4 (-15.63%)	1051.56 (-7.24%)	332.76 (-12.86%)	391.56 (-3.21%)
8x8	Hybrid	839	1900.32	2264.96	736.56	805.92
	Optimized-Hybrid	783 (-7.15%)	1656 (-14.75%)	2114.96 (-7.09%)	640.96 (-14.91%)	787.28 (-2.37%)
16x8	Hybrid	458	2117.76	4624.48	1506.72	1621.28
	Optimized-Hybrid	422 (-8.5%)	1789.28 (-18.35%)	4240 (-9.1%)	1379.04 (-9.25%)	1597.28 (-1.5%)
32x8	Hybrid	261	2411.84	9241.28	2900.16	3204.16
	Optimized-Hybrid	246 (-6.1%)	2055.36 (-17.34%)	8355.52 (-10.6%)	2694.08 (-7.65%)	3116.16 (-2.82%)
64x8	Hybrid	151	2693.12	17834.88	5703.04	6366.08
	Optimized-Hybrid	145 (-4.14%)	2318.72 (-16.15%)	15992.96 (-11.52%)	4753.28 (-19.98%)	6273.28 (-1.48%)

Fig. 4. Performance, Power and Energy Consumption Comparison.

As we discussed before, the hybrid earthquake application is memory bound. From our experiments, we found that L2 cache behavior is a major factor for memory power consumption. So additional loop optimizations such as using loop blocking with a block size of 8x8 and unrolling nested loops four times are applied to the application in order to improve cache utilization.

Figure 4 shows the performance, power, and energy consumption comparison. 2x8 stands for 2 nodes with 8 cores per node. For the hybrid application executions, we use 1 MPI process per node with 8 OpenMP threads per node as default. Optimized-Hybrid stands for applying DVFS, DCT and loop optimization to the hybrid application for execution. System Energy means the total energy consumed by the number of nodes used, and its unit is kilojoule. System Power means the total power consumed by the number of nodes used. CPU Power means the total power consumed by all CPUs used. Memory Power means the total power consumed by all memory components used. Overall, we reduce the execution time and lower power consumption of the hybrid application by applying DVFS (MPI), DCT (OpenMP) and loop optimizations. This is a good improvement in performance and power consumption.

As shown in Figure 4, the Memory Power consumption is larger than the CPU Power consumption for the hybrid earthquake application because the application is memory-bound, and applying DVFS (MPI), DCT (OpenMP) and loop optimizations to the hybrid application benefit the CPU power consumption more than the Memory power consumption because we used 2 OpenMP threads per node when applying DCT to the hybrid application. It means that there are 6 idle cores per node during the execution of the dominated function qdct3 and hourglass. This results in a big CPU power saving. However, loop optimizations just improve cache utilization a little bit. For instance, on 512 cores (64x8), the optimized hybrid application execution results in 4.14% improvement in Runtime, 16.15% improvement in System Energy consumption, 11.52% improvement in System Power consumption, 19.98% improvement in CPU Power consumption, and 1.48% improvement in Memory Power consumption. Overall the System Energy got the best improvement percentage because the System Energy is the product of Runtime and System Power and both Runtime and System Power got improved. Overall, applying DVFS (MPI), DCT (OpenMP) and loop optimizations to the hybrid MPI/OpenMP earthquake application reduced up to 8.5% the execution time and lowered up to 18.35% energy consumption of the hybrid application.

V. Summary

In this paper, we presented the MuMMI framework which consists of MuMMI Instrumentor, Databases and Analyzer in detail. As part of MuMMI project, we mainly discussed a MuMMI instrumentor to provide automatic performance and power data collection and storage with low overhead on the multicore system SystemG in detail, then utilized the MuMMI Instrumentor to collect performance and power data for the hybrid MPI/OpenMP earthquake application to discuss the application performance-power trade-off and optimizations. Our experimental results shows that we reduced up to 8.5% the execution time and lowered up to 18.35% energy consumption of the hybrid application by applying DVFS (MPI), DCT (OpenMP) and loop optimizations. Future work will extend our MuMMI work to further address power and energy consumption issues in addition to application performance on petascale systems with GPUs, and plan to develop a web-based automated performance, power and energy modeling system to provide automatic performance, power and energy modeling online as we did for Prophesy system in [22].

VI. Acknowledgments

This work is supported by NSF grants CNS-0911023, CNS-0910899, and CNS-0910784. We would like to thank Dr. Benchun Duan from Department of Geology & Geophysics, Texas A&M University for providing the dataset for TPV210.

References

[1] S. Browne, et al., "A portable programming interface for performance evaluation on modern processors," International Journal of High-Performance Computing Applications 2000.

[2] M. Curtis-Maury, J. Dzierwa, et al., "Online Power-Performance Adaptation of Multithreaded Programs using Hardware Event-Based Prediction", the International Conference on Supercomputing (ICS06), 2006.

[3] R. Ge, X. Feng, and K. W. Cameron, "Performance-constrained Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters." In IEEE/ACM SC 2005, Seattle, WA, 2005.

[4] R. Ge, X. Feng, S. Song, et al., "PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications", IEEE Trans. on Parallel and Distributed Systems 21(5): 658-671, 2010.

[5] Integrated Performance Monitoring (IPM), http://ipm-hpc.sourceforge.net/

[6] N. Kappiah, V. Freeh, and D. Lowenthal. "Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs", the 2005 ACM/IEEE Conference on Supercomputing (SC05), 2005.

[7] C. Lively, X. Wu, V. Taylor, S. Moore, H. Chang, C. Su and K. Cameron, "Power-Aware Predictive Models of Hybrid (MPI/OpenMP) Scientific Applications on Multicore Systems", International Conference on Energy-Aware High Performance Computing (EnA-HPC2011), Hamburg, Germany, September 7-9, 2011.

[8] C. Lively, X. Wu, V. Taylor, S. Moore, H. Chang, and K. Cameron, "Energy and Performance Characteristics of Different Parallel Implementations of Scientific Applications on Multicore Systems", International Journal of High Performance Computing Applications (IJHPCA), Volume 25 Issue 3, August 2011, pp. 342 - 350.

[9] C. Lively, E-AMOM: An Energy-Aware Modeling and Optimization Methodology for Scientific Applications on Multicore Systems, Ph.D Dissertation, Department of Computer Science & Engineering, Texas A&M University, May 2012.

[10] Multiple Metrics Modeling Infrastructure (MuMMI) project, http://www.mummi.org

[11] PAPI (Performance Application Programming Interface), http://icl.cs.utk.edu/papi/

[12] Score-P, Scalable Performance Measurement Infrastructure for Parallel Codes, http://www.vi-hps.org/projects/score-p/

[13] The SCEC/USGS Spontanous Rupture Code Verification Project, http://scecdata.usc.edu/cvws.

[14] TAU (Tuning and Analysis Utilities), http://www.cs.uoregon.edu/research/tau/home.php

[15] V. Taylor, X. Wu, J. Geisler, and R. Stevens, "Using Kernel Couplings to Predict Parallel Application Performance." In Proc. of the 11th IEEE International Symposium on High-Performance Distributed Computing (HPDC 2002), Edinburgh, Scotland, July 24-26, 2002.

[16] V. Taylor, X. Wu, and R. Stevens, "Prophesy: An Infrastructure for Performance Analysis and Modeling of Parallel and Grid Applications," ACM SIGMETRICS Performance Evaluation Review, vol. 30, no. 4, March, 2003.

[17] X. Wu, V. Taylor, and R. Stevens, "Design and Implementation of Prophesy Automatic Instrumentation and Data Entry System." In the 13th International Conference on Parallel and Distributed Computing and Systems (PDCS2001), Anaheim, CA, August, 2001.

[18] X. Wu, B. Duan, and V. Taylor, "Parallel Earthquake Simulations on Large-scale Multicore Supercomputers" (Book Chapter), Handbook of Data Intensive Computing (Eds: B. Furht and A. Escalante), Springer-Verlag, 2011.

[19] X. Wu, B. Duan, and V. Taylor, "Parallel Finite Element Earthquake Rupture Simulations on Quad- and Hex-core Cray XT Systems", the 53rd Cray User Group Conference (CUG2011), May 23-26, 2011, Fairbanks, Alaska.

[20] X. Wu, V. Taylor, J. Geisler, and R. Stevens, "Isocoupling: Reusing Coupling Values to Predict Parallel Application Performance," in 18th International Parallel and Distributed Processing Symposium (IPDPS2004). Santa Fe, New Mexico, 2004.

[21] X. Wu, et al., "Design and Development of Prophesy Performance Database for Distributed Scientific Applications." In Proc. the 10th SIAM Conference on Parallel Processing for Scientific Computing, Virginia, 2001.

[22] X. Wu, V. Taylor, and J. Paris, "A Web-based Prophesy Automated Performance Modeling System", the International Conference on Web Technologies, Applications and Services (WTAS2006), July 17-19, 2006, Calgary, Canada.

[23] V. Weaver, PAPI overhead on various kernel interfaces, http://web.eecs.utk.edu/∼vweaver1/projects/papi-cost/