MAximum Multicore POwer (MAMPO) - An Automatic Multithreaded Synthetic Power Virus Generation Framework for Multicore Systems

Karthik Ganesan and Lizy K John
ECE Department, University of Texas at Austin, TX, USA
karthik@mail.utexas.edu and ljohn@ece.utexas.edu

ABSTRACT

The practically attainable worst case power consumption for a computer system is a significant design parameter and it is a very tedious process to determine it by manually writing high power consuming code snippets called power viruses. Previous research efforts towards automating the power virus generation process are all limited to the single core processors and are not effective when applied to multicore parallel systems as the components like the interconnection network, shared caches, DRAM and coherence directory also contribute significantly to the power consumption of a multicore parallel system. In this paper we propose MAximum Multicore POwer (MAMPO), which is the pioneer attempt towards a framework to automatically generate a multithreaded power virus for a given multicore parallel system configuration. We show that the the power viruses generated by MAMPO consume 40% to 89% more power than running multiple copies of single-core power viruses like MPrime torture test and the most recent published previous work called SYMPO on 3 different parallel multicore system configurations. The superiority of the MAMPO viruses are also shown by comparing the power consumption of the MAMPO viruses with that of the workloads in the PARSEC benchmark suite and that of the commercial Java benchmark SPECjbb. The MAMPO viruses consume 45% to 98% more power than that of the average power consumption of the workloads in the PARSEC suite and 41% to 56% more power than that of the commercial benchmark SPECjbb.

1. INTRODUCTION

Due to power delivery, thermal and cooling issues along with a world-wide initiative towards green computing, power consumption is a first class design parameter in high end server systems and it has always been a significant constraint in low end embedded system design. More specifically, the maximum power consumption for which computer systems are designed, called the Thermal Design Power (TDP) is one of the most important of the different design parameters and is something that is very carefully determined by the computer architects. The cooling systems of these modern processors/memories are designed in such a way, that these systems are deemed to safely operate only within this power cap and are equipped with the capability to automatically throttle down the operating frequency when the system is driven to reach this maximum power. This maximum power consumption for which a system is designed cannot just be fixed as the sum of the power consumption of the various components in the system, but rather it has to be the maximum attainable power consumption that a user workload could practically achieve in the system under design. This is due to the fact that this maximum attainable power consumption is quite low compared to the sum of the power consumption of various micro-architectural components as it is almost impossible to keep all these components of a system simultaneously active by any workload. The process of determining the maximum power for a design is very complicated due to it's dependence on multiple factors like the workload that could be executed, the configuration of the system, the power saving features implemented in hardware and the way some of these features are exercised by the operating system.

If the maximum power of a design is fixed too high, a designer will end up wasting a lot of resources by over-provisioning the heat sinks, cooling system, power delivery system and various other system level power management utilities. A related example will be the design of external power supplies to server systems. Due to incognizance of the precise maximum attainable power of a system, a power supply could be designed to handle a high load and when the typical usage scenario is far below that load, the efficiency of the power supply is known to drop many folds [1]. It is to be noted that over provisioning of these power related utilities could result in substantial increase in maintenance costs of servers as it is estimated that for every watt of power used by the computing infrastructure in a data center, another 0.33 to 0.5 watt of power is required by the cooling system [2] [3] due to the ongoing rack-level compaction [4]. On the other hand, if this maximum power consumption is underestimated, it results in affecting the overall system reliability and availability due to overheating. When the ambient temperature increases beyond the safe operating limits, it could result in early failure of the micro-architectural components resulting in sporadic system freezes and crashes.

In an effort towards fixing the maximum power consumption of systems at the most optimal point, architects are used to hand-crafting possible code snippets called power viruses [5] [6]. But, this process of trying to manually write such maximum power consuming code snippets is very tedious [7]. This tedium is due to the fact that there are so many components that interact when a workload executes on a processor/system making it intractable to model all these complex interactions and requires a profound knowledge about these interactions to be able to write a code snippet that will exactly exercise a given execution behavior. Adding to this complexity are the various power saving features implemented in

SC11, November 12-18, 2011, Seattle, Washington, USA

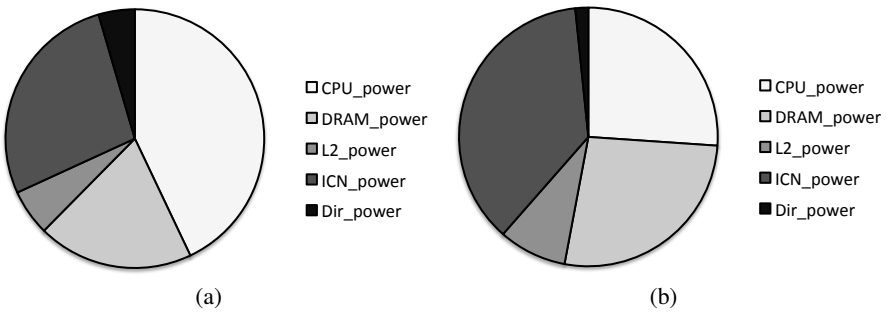


Figure 1: Breakdown of power consumption of the PARSEC benchmark *fluidanimate* in (a) System with eight 4-wide out-of-order cores, 4MB L2, 8GB DRAM and (b) System with sixteen 2-wide out-of-order cores, 8MB L2, 16GB DRAM

the hardware like clock gating, demand based switching, enhanced speed step technology and the various power states of the CPUs exercised by the operating system. Lastly, one cannot be sure that the manually written power virus is the practically possible maximum case to be able to safely design the processor for this particular maximum power. As a result of this, designers tend to end up in the aforementioned wasteful over-provisioning.

Cognizant of the significance of this problem, there has been some recent efforts by Ganesan et. al [8] and Joshi et. al [9] towards automating the generation of power viruses using machine learning. But both of the previous work are limited to the power consumption of single-cores and do not address the complexities involved in generating a power virus for a modern multicore parallel system. It is to be noted that there are many components like the interconnection network, shared caches, memory subsystem and cache coherence directory other than the CPU that significantly contribute to the overall power consumption of a multicore parallel system. Figures 1(a) and 1(b) show the breakdown of power consumption of a randomly chosen PARSEC [10] benchmark *fluidanimate* on two typical modern multicore systems with eight and sixteen cores respectively. The eight core system has eight 4-wide out-of-order cores with 4MB L2 and 8GB DRAM and the sixteen core system has sixteen 2-wide out-of-order cores with 8MB L2 and 16GB DRAM. One can see that the total power consumption of all the cores sum up to only 41% and 21% of the whole system power for the oct-core and sixteen-core systems. It is found that running multiple copies of these single-core power viruses (as in previous work [8] or MPrime [11]) on multiple CPUs of a multicore parallel processor is not even close to the power consumption of a power virus generated specifically for a given multicore parallel system. This is due to fact that such a single-core power virus like MPrime is very compute-bound lacking in data movement resulting in a reduced activity in the shared caches and the interconnection network. Due to upcoming memory hungry technologies like virtualization, the continuously more memory-seeking nature of today's search and similar Internet based applications along with a shift in paradigm from multicore to many-core, we see that only the power levels of processors being controlled and capped, while we do not see any signs of slow down in the increase in power consumption of memory and interconnects making it more important to be aware of their worst-case power characteristics.

In this paper we propose MAximum Multicore POwer (MAMPO), which consists of a multithreaded synthetic workload generator driven by machine learning aimed at automatically finding the best power virus for a given multicore parallel system configuration in the pre-silicon design stage of a system. This is the first attempt towards answering many questions about how to efficiently search for a power virus for multicores viz., i) which are the most important dimensions of the abstract workload space that should be modeled for a multicore system, ii) what is the required amount of granularity in each dimension and especially the detail at which the core level out-of-order execution related workload characteristics should be modeled iii) if it is worthwhile to make the threads heterogeneous and deal with state space explosion problem or should the threads be homogeneous iv) what are the data sharing patterns (producer-consumer, migratory etc) that should be exercised to stress the interconnection network, shared caches and DRAM effectively, and many other similar questions, each of which are further elaborated later in this paper. The major contributions of this paper are,

- Proposal of MAximum Multicore POwer (MAMPO), which is the pioneer attempt towards a multithreaded synthetic power virus generation framework targeting multicore systems.
- Validation of MAMPO by comparing the power consumption of the generated multithreaded virus with that of running multiple copies of the industry grade power virus MPrime torture test [11] and SYMPO [8] on multiple cores for three different parallel system configurations. We also compare the power consumption of the MAMPO viruses with that of the commercial Java benchmark SPECjbb. The MAMPO virus consumes 40% to 89% more power than the parallel run of single-core viruses and 41% to 56% more power than that of SPECjbb for three parallel system configurations studied.
- Further validation of MAMPO by comparing the power consumption of the generated power viruses with that of the workloads in the PARSEC benchmark suite. We show that MAMPO virus consumes 45%, 52% and 98% more power than the maximum power consuming benchmark in the PARSEC [10] suite for the 3 parallel system configurations studied.

The rest of the paper is organized as follows: In Section 2, we introduce MAMPO, our multithreaded power virus generation framework targeting multicore parallel system and Section 3 elaborates on the three different parallel system configurations used to validate MAMPO along with the results showing the efficacy of the generated power viruses. We provide related work in Section 4 and conclude in Section 5.

2. MAMPO

A power virus for a muticore system has to stress different parts of the system in such a way that the overall power consumption is maximized. As we mentioned already, keeping all the components

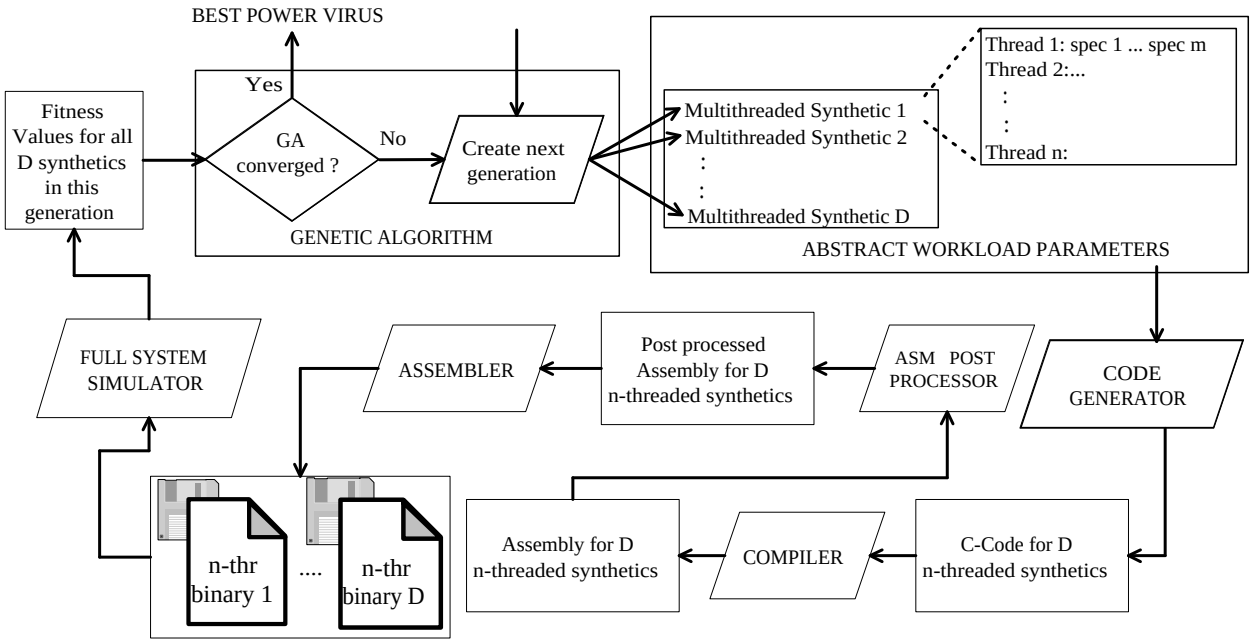


Figure 2: Multithreaded power virus generation framework

of a system simultaneously active is not possible. For example, to be able to stress the DRAM of a system, the processor may have to stall for many cycles for those long-latency loads to complete. Any program that is completely memory-bound cannot consume much power in the cores and a program that is completely compute-bound cannot consume much power in the memory, caches or the interconnection network. Thus, the power virus has to strike the right balance between stressing different power consuming components in the system to be able to maximize the overall power. There are many latency hiding mechanisms implemented throughout a modern computer system starting from the out-of-order execution circuitry in the cores, various buffers, the miss status handling registers in the caches, pipelining in the interconnection network to various optimizations implemented in the DRAM controller and all of these numerous features should be exploited to the right extent by this power virus to achieve maximum overall power. Mainly to avoid the need to model all these complex interactions, we use a black-box approach that employs a machine learning based search technique along with a multithreaded workload generator to automatically search for a power virus given a multicore system configuration.

The main components of the MAMPO framework are, i) the abstract workload model used ii) the code generator, compiler and the assembly post processor iii) the full system simulator with detailed power models used to evaluate the power consumption of the multithreaded synthetics and iv) the machine learning technique employed in the framework, Genetic Algorithm (GA). Figure 2 shows the flowchart of the power virus generation framework. The Genetic Algorithm (GA) [12] generates the parameter values for the potential candidates for the synthetic power virus case as it iteratively searches through the abstract workload space. These generated abstract workload characteristics are fed to the code generator that generates a multi-threaded synthetic C program containing embedded assembly instructions for each thread based on these specified characteristics. This multi-threaded C code is then translated to direct assembly code with the help of a compiler. At times, the compiler introduces some spurious stack operations amidst the set of instructions that we incorporate as embedded assembly and this assembly code has to be post processed to remove such unnecessary instructions and then it is further compiled into a SPARC binary. This multi-threaded SPARC binary is then executed on a full system simulator with cycle accurate power models for various system components to evaluate the power consumption of the generated synthetic on the system configuration under study. These power consumption numbers are fed back to the Genetic Algorithm to intelligently choose the next set of potential candidates for the power virus and this process iteratively continues until the search converges to find the best power virus for a given system configuration. Each of the components of this framework will be explained in detail further in this Section.

2.1 Genetic Algorithm

The machine learning approach we use in our framework is popularly called the Genetic Algorithm (GA) [12]. GA is a search technique inspired by evolutionary biology where problem solutions are encoded as chromosomes and these chromosomes are mutated and recombined to form newer chromosomes. A population in the genetic space is defined as a set of chromosomes or possible outcomes of the problem under investigation. The algorithm proceeds by first choosing a set of D random chromosomes as the initial population, where D is the deme size or the population size used in the algorithm. For our power virus search, a chromosome will refer to the set of parameters in the abstract workload space for a candidate multithreaded synthetic workload. These D random chromosomes (multithreaded synthetics) form the first generation of individuals for the algorithm to get started. These synthetics of the first generation are evaluated for their fitness, which is their overall system level power consumption in our problem on the system configuration under study. The fitness values represent the quality of these individuals in the population and are fed back to the GA. Based on the fitness values of these synthetics, there are different operators that are applied on them like mutation, crossover and elite repro-

duction to produce the chromosomes of the individuals of the next generation, which are again evaluated for their fitness and fed back to the GA. This evolutionary process continues until the Genetic Algorithm converges with the same value for each of the different dimensions and is repeated by seeding the GA with different random seeds to make sure that the results are robust. Though one may argue that this process of GA does not necessarily guarantee to achieve the best theoretically maximum power virus as it is still a heuristic based global optimization technique, by seeding the GA with different starting points and running it until convergence does guarantee a tight upper-bound for the maximum power for practical purposes.

The most significant operators used in GA are mutation and crossover. Mutation operator probabilistically chooses parts of the chromosome and modifies them to form new chromosomes. In our case, the specifications of the multithreaded synthetics in terms of abstract workload parameters are modified randomly to form new multithreaded synthetics. The crossover operator recombines parts of two chosen chromosomes in some way to form a new chromosome for the offspring. The specifications of two chosen multithreaded synthetics are combined in a meaningful way to form the specifications of the new multithreaded synthetic offspring. We will further explain the values used for these GA parameters like the mutation rate, crossover rate and reproduction rate for our power virus search problem in Section 3.

In this Subsection, we further explain why we chose GA over other search techniques. Firstly, as a general rule of thumb, a directed search technique like Genetic Algorithm (GA) is more efficient than a random search technique or a brute force methodology. Through various experiments, we have found that the crossover operator employed in GA is very effective when searching through the workload space for a power virus. This is because when we cross over two good solutions in our space, the characteristics of the parents can be very meaningfully merged and hence the offspring is also usually good, when compared to a random sample in the same space. In the rest of this Section, we elaborate on the abstract workload model that is employed and the process of code generation for the multithreaded synthetic workload.

2.2 Abstract Workload Model

The effectiveness of this kind of power virus generation framework lies mainly in the efficacy of the abstract workload space that is being searched through by machine learning. Firstly, the dimensions of this abstract workload space should be as much microarchitecture independent as possible to enable this framework to be able to generate the best power virus for different types of microarchitectures. It is the job of the machine learning algorithm to take care of tailoring the parameters of the abstract workload model to maximize the power consumption for a given microarchitecture based on power estimates provided by the simulator for this microarchitecture under study. But, it is also important that these dimensions of the abstract workload space be robust enough to be able to vary the execution behavior of the generated workload in every part of the multicore system. It is to be noted that the dimensions should also not be too many as it could also result in a situation where the search would never converge due to a state space explosion. The characteristics of real-world programs that affect performance and in turn the power consumption are carefully studied and is used as a guide to design these dimensions as it is important that the generated power virus should still be a realistic workload depicting the practically attainable maximum power. In earlier approaches for synthetic benchmark generation for uniprocessors, researchers came up with metrics to characterize the execution behavior of programs on single core processors [13] [14] [15] [16] [17]. In this paper, we come up with similar metrics for the generation of multithreaded synthetics for multicore systems.

In the abstract workload model, we have the choice of searching for a multithreaded power virus with homogeneous thread characteristics or provide the GA with the flexibility to configure the threads to be heterogeneous. It is to be noted that, when the threads are made heterogeneous, almost we multiply the number of dimensions in the abstract workload space for every thread by the number of threads. This could possibly result in a state space explosion and the GA may never converge. But, on the other hand, most of the real world parallel applications have heterogeneous thread characteristics [10] at least in their data access pattern. For example, one of the most commonly used data access pattern is the producer-consumer relationship between simultaneously executing threads, where one or more producer threads write data, which is read by one or more consumer threads. To be able to exercise such a behavior in the synthetic, there should be some amount of heterogeneity in the threads to be able to act as a producer and a consumer thread. At a minimum, there should be some heterogeneity in the instruction mix in terms of the number of loads or stores. But, due to this heterogeneity in the instruction mix, the other core-level dimensions may also need be adjusted heterogeneously to be able to consume maximum power. For example, the producer threads may need to have a different register dependency distance or branch predictability than the consumer thread to be able maximize the power consumption of the core, in turn to keep the system at its maximum attainable power. Figure 3 shows the different dimensions of our abstract workload model and their granularity. Further in this Section, we explain each of these dimensions or what we call as the 'knobs' of our workload generator. We first begin by explaining the intuition behind the design of this abstract workload space.

In our abstract workload model, we have a controlled amount of heterogeneity, where only a few heterogeneous classes of threads can be configured and all the threads in the system have to belong to one of these few heterogeneous classes. The threads within a class are homogeneous. This controls the state space explosion and we will also be able to mimic the communication characteristics of the real parallel applications. We have found that a reasonable number for heterogeneous classes is four, up to which the state space is tractable and also allows to control power for the major power consuming components. Investigation in previous research [18][19][20][21][22] about the communication characteristics of the parallel applications has showed that there are four significant data sharing patterns that happen in real parallel applications, namely,

1. **Producer-consumer sharing pattern**: One or more producer threads write to a shared data item and one or more consumers read it. This kind of sharing pattern can be observed in the SPLASH-2 benchmark *ocean*.

2. **Read-only sharing pattern**: This pattern occurs when the shared data is constantly being read and is not updated. SPLASH-2 benchmark *raytrace* is a good example exhibiting this kind of a behavior.

3. **Migratory sharing pattern**: This pattern occurs when a processor reads and writes to a shared data item within a short period of time and this behavior is repeated by many processors. A good example of this behavior will be a global counter that is incremented by many processors.

4. **Irregular sharing pattern**: There is not any regular pattern into which this access behavior can be classified into. A good

#	Knob name	Knob range	# Thrd classes	Category
1	Number of threads	1, 4, 8, 16, 32	4	Parallelism
2	Thread class & processor assignment	1, 2, ... 12	-	Shared data access pattern and communication characteristics
3	Percent memory accesses to shared data	10, 30, 50, 60, 70, 90	4	
4	Shared memory access strides in two buckets	0, 4, 8, 12, 16, 32, 64	4	
5	Coupled load-stores	True/false	1	
6	Private memory access strides in two buckets	0, 4, 8, 12, 16, 32, 64 in each bucket	4	Private data access pattern
7	Working set size (# loop iterations before array ptr. reset)	1, 10, 20, 40, 100, 200	4	Memory footprint
8	Memory Level Parallelism (MLP)	1, 2, 3, 4, 6	4	Memory level parallelism
9	MLP frequency	High, low	1	
10	Average basic block size	10, 20, 30, 50, 100	1	Control flow predictability
11	Average branch predictability	0.8, 0.86, 0.92, 0.96, 0.98, 0.99, 1.0	4	
12	INT ALU proportion	0 - 4	4	Instruction mix
13	INT MUL proportion	0 - 4	4	
14	INT DIV proportion	0 - 4	4	
15	FP ADD proportion	0 - 4	4	
16	FP MUL proportion	0 - 4	4	
17	FP DIV proportion	0 - 4	4	
18	FP MOV proportion	0 - 4	4	
19	FP SQRT proportion	0 - 4	4	
20	LOAD proportion	0 - 4	4	Instruction mix, data access pattern and communication characteristics
21	STORE proportion	0 - 4	4	
22	Register dependency distance (number of instructions)	1, 2, 4, 8, 16, 32, 64	4	Instruction level parallelism
23	Random seed	1, 2, 3	1	Code alignment

Figure 3: Abstract workload model

example will be a global task queue, which can be enqueued or dequeued by any processor which does not follow a particular order.

Though the above said patterns are the most commonly occurring sharing patterns, subtle variations of each one or more than one sharing pattern may be occurring in a multicore system. In our framework, we use a generic memory access model, which when parameterized accordingly, can yield any combination of the above said sharing patterns. In our abstract workload model, we do not include some characteristics of parallel applications like locks and barriers, because the presence of locks and barriers always result in slowing down the execution of applications resulting in a lower overall power consumption. Next in this Section, we provide a brief overview of our generic memory access model and then elaborate on each of the different dimensions of the abstract workload space or what we call as the 'knobs' of our workload generator.

Our memory access model is mainly based on a 'stride' based access pattern [13] in terms of static loads and stores in the code. When profiling a modern workload, one can observe that each of the static loads/stores access the memory like in an arithmetic progression, where the difference between the addresses of two successive accesses is called the stride. It is known that the memory access pattern of most of the SPEC CPU2000 and the SPEC CPU2006 workloads can be safely approximated to be following a few dominant stride values [15] [14]. In our abstract workload model, we handle the stride values of the memory accesses to the private and shared data separately. For both of the shared and the private memory accesses, the stride values are grouped into two bins and the stride value assigned to a memory access instruction is chosen with equal probability from each of the bins. Along with the stride access patterns, the proportion of loads and stores in each thread also affect the data sharing pattern of the synthetic workload. For example, to achieve the producer-consumer sharing pattern between two threads, one will have to configure the instruction mix in such a way that the loads to shared data in the consumer and the stores to shared data in producer are in the right proportion and also configure the remaining knobs like the percent memory accesses to shared data, strides to shared data, thread assignment to processors and working set size to enable these threads to communicate the right amount of data between each other in a given pattern. Though our model is robust enough to model parallel applications and their behavior, it can also be configured to model loosely related threads of commercial applications by making the 'percent shared' knob to be very low.

The branch predictability of the benchmark can be captured independent of the microarchitecture by using the branch transition rate [23]. The branch transition rate captures the information about how quickly a branch transitions between taken and not-taken paths. A branch with a lower transition rate is easier to predict as it sides towards taken or not-taken for a given period of time and rather a branch with a higher transition rate is harder to predict. First, the branches that have very low transition rates, can be generated as always taken or always not taken as they are easily predictable. The rest of the branches in the synthetic need to match the specified distribution of transition rate, which is further explained in the next Subsection. We provide a brief description about each of the different knobs of our workload generator:

1. **Number of threads**: The number of threads knob controls the amount of thread level parallelism of the synthetic workload. This varies from only one thread up to 32 threads executing in parallel.

2. **Thread class and processor assignment**: This knob con-

trols the thread classes to which each thread gets assigned. Up to 12 patterns are used to model the combinations of these assignments. These combinations also include various permutations in terms of how far/near the threads of same/different classes are placed based on the assignment to the processors on which they are bound to execute.

3. **Percent memory accesses to shared data**: This knob controls what proportion of memory accesses are to the shared data and the rest of the memory accesses are directed to private data. This knob can be configured separately for each thread class to allow the sharing percentage to be heterogeneous across thread classes. This heterogeneity may help the threads to be configured to differently stress the private and shared caches.

4. **Shared memory access stride values**: As mentioned earlier, two bins of stride values are specified for the shared memory accesses and every such memory access can be configured to have any one of the two bins with equal probability. This knob can also be configured separately for each of the different threads, to be able to allow each one of them to uniquely stress differ levels in the memory hierarchy.

5. **Coupled load-stores**: When this knob is set to true, an effort is made to couple a load with a following store to be able to mimic a migratory sharing pattern of access. This migratory sharing pattern can create huge amounts of traffic when coherence protocols like MESI is used where there is not a specific state for a thread to own the data.

6. **Private memory access stride values**: Similar to the stride values to the shared memory, two bins of stride values are specified for the private memory accesses and every such memory access can be configured to have the stride from any one of the two bins with equal probability. This knob can also be configured separately for each thread class to be able to stress different levels of the memory hierarchy separately.

7. **Working set size**: This knob controls the working set size of the synthetic. The correspondence of this knob to the real implementation in terms of number of iterations of one of the nested loops in the synthetic will be explained in detail in the next Subsection. This knob can be configured separately for different thread classes to be able to allow various cache resource sharing patterns in terms of varying working set sizes.

8. **Memory Level Parallelism (MLP)**: This knob controls the amount of Memory Level Parallelism (MLP) in the workload, which is defined as the number of memory operations that can happen in parallel and is typically used to refer to the number of outstanding cache misses at the last level of the cache. The number of memory operations that can occur in parallel is controlled by introducing dependency between memory operations. This knob can also be configured separately for every thread class to enable the threads to have various access patterns to the DRAM.

9. **MLP frequency**: Though the MLP knob controls the burstiness of the memory accesses, we need one more knob to control how frequently these bursty behaviors happen.

10. **Average basic block size**: This refers to the average number of instructions in a basic block in the generated embedded assembly based synthetic code. This knob is specified to be homogeneous across different thread classes.

11. **Branch predictability**: The branch predictability of a workload is an important characteristic that also affects the overall throughput of the pipeline. When a branch is mispredicted, the pipeline has to be flushed and this results in a reduced activity in the pipeline.

12. **Instruction mix**: The Instruction mix is decided based on the proportions of each of the instruction types INT ALU, INT MUL, INT DIV, FP ADD, FP MUL, FP DIV, FP MOV and FP SQRT. The GA can configure each of these weights associated with the instruction types to be anything between zero to four, controlling the proportion of each instruction type. Some restrictions are placed on the instruction mix by writing rules in the GA like a minimum number of INT ALU instructions should be present if there are any memory operations in the code to be able to perform the address calculation for these memory operations.

13. **Register dependency distance**: This knob refers to the average number of instructions between the producer and consumer instruction for a register data. If the register dependency distance is high, the Instruction Level Parallelism (ILP) in the synthetic is high resulting in a high activity factor in the pipeline of the core. But, if the register dependency distance is low, the out-of-order circuitry like the ROB and other buffers may have higher occupancy resulting in a higher activity factor in these parts of the core. This knob is required to be configured separately for different thread classes, as different threads having different memory latencies may need to have different amounts of ILP to maximize the power consumption in the cores.

14. **Random seed**: This knob controls the random seed that is used as an input to the statistical code generator, which will generate different code for the same values for all the other knobs. It mostly affects the alignment of the code or the order in which the instructions are arranged. This order of instructions does have some impact on maximizing the power and we include this knob also to be explored by the GA in the search for a power virus.

2.3 Code Generation

In this Section we explain how the final code generation happens based on the knob settings given in terms of the abstract workload parameters. Figure 4 shows an overview of code generation. The generated code consists of the main function and a function for each thread that is spawned from the main function using the pthread_create() system call. The required amount of shared data is declared and allocated in the main function as a set of integer/floating point arrays and the pointers to these arrays are available to each of the threads. The private data that is supposed to be used by every thread is declared and allocated within the function for each thread. Each of the threads also bind themselves with the processor number specified when the code was generated based on the thread class and processor assignment knob. We use a barrier synchronization to synchronize all the threads after they finish their respective system calls for allocating their private data arrays and binding themselves to the assigned processor.

The body of each thread consists of two inner loops filled with embedded assembly and one outer loop encompassing these inner loops. As previously mentioned, our memory model is a stride based access model, where the loads and stores in the generated synthetic access the elements of the private/shared arrays, each static load/store with a constant stride. The address calculation for

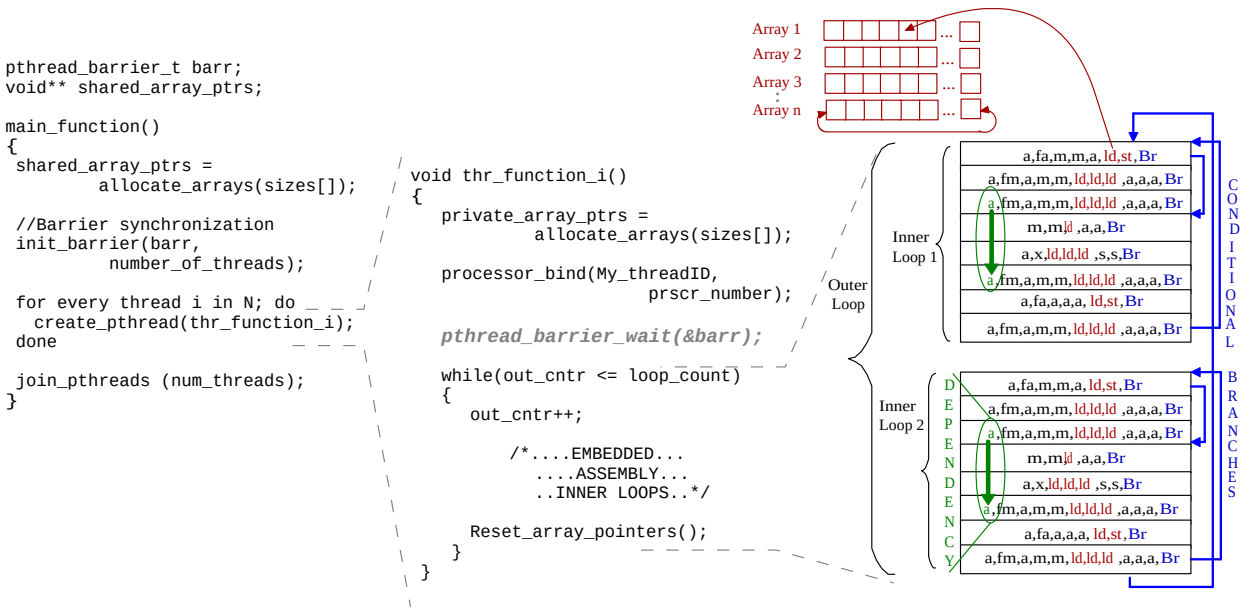


Figure 4: Multithreaded synthetic workload generation

the next access of each load/store is done by using other ALU instructions in the generated code for each of the array pointers by using the assigned stride value. When the specified working set size is covered, the pointers that are used are reset to the beginning of the array. This pointer reset is done outside the inner loops and inside the encompassing outer loop enabling us to control the working set size with the number of iterations of the inner loop and the number of dynamic instructions with the number of iterations of the outer loop. The embedded assembly contents of the two inner loops are the same if the MLP frequency knob is set to high. If the MLP frequency knob is set to low, the memory operations in the second loop are removed so that the bursty memory access behavior happening in the first loop occurs at a lower frequency.

The required branch predictability or the control flow behavior in the synthetic is achieved by grouping branches into pools with each pool assigned to a condition register. The branches are taken/not taken based on whether this assigned condition register is set or not. This condition register is set/unset by using a modulo operation on the control variable of the loop. The only information that is required to generate the main function is the biggest shared data footprint amongst the different threads to be able to allocate the shared arrays. The following steps are followed to generate the code for every thread based on the corresponding knob settings for each:

1. Generate the code to allocate the required amount of space for private data accesses based on the percent shared accesses, proportion of memory operations in instruction mix and the working set size.

2. Generate the processor_bind() system call using the assigned processor number and then a barrier synchronization system call is generated.

3. Generate the code for outer-loop based on the dynamic number of instructions desired taking into account the average basic block size and the number of basic blocks.

4. Fix the code spine for the first inner loop based on a fixed number of basic blocks and the average basic block size knob.

5. For each of the basic block in the first inner loop, configure the instruction type of each instruction by stochastically choosing from the instruction mix information. If the coupled load-stores is true, the instructions are swapped based on a bubble sort fashion in such a way that a store is made to follow a load and they are made to access the same address.

6. The number of branch groups and the modulo operation are fixed based on the required average branch predictability. The modulo operation for each of the branch groups are generated at the beginning of the inner loop based on the loop count and a register is set/unset to decide if those branches for this particular group are going to be taken or not taken for this loop iteration. Branches are generated to fall through or take the target to another basic block based on their assigned register value.

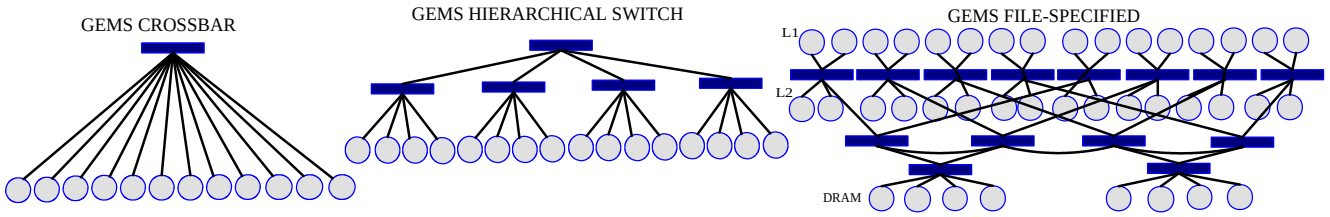
7. Using the average dependency distance knob, each of the operands of every instruction is assigned with a previous producer instruction. Some rules are used to check the compatibility between producer and the consumer in terms of the data that is produced by the producer instruction and that consumed by the consumer. If two instructions are found to be not compatible, the dependency distance is incremented or decremented until a matching producer is found for every instruction. The memory level parallelism information is also used to assign load-load dependencies in this process.

8. Based on the percent shared accesses knob, each of the memory operations are classified into the ones that access shared data and the ones that access private data. Based on the stride value of the corresponding memory operation (shared or private and based on the assigned bin), their corresponding address calculation instructions are given the stride values as immediate operands.

9. Register assignment happens by first assigning the destination registers in a round robin fashion. The source register

Parameter	System – I	System – II	System – III
No. of cores	4	8	16
DRAM	4 GB	8 GB	16 GB
L1 cache	64 KB, 4 way, 2 cycles	32 KB, 4 way, 1 cycle	16 KB, 2 way, 1 cycle
L2 cache	4 MB, 4 way, 4 banks	4 MB, 8 way, 8 banks	8 MB, 16 way, 16 banks
L1, L2 MSHRs	48	32	24
ROB	128	64	32
Mach-width	8	4	2
Branch pred.	YAGS, 12 bit PHT	YAGS, 11 bit PHT	YAGS, 10 bit PHT
BTB size	1024	512	256
Int ALUs	4 ALU, 2 Int div	3 ALU, 1 Int div	2 ALU, 1 Int div
Topology	Crossbar	Hierarchical switch	File-specified
FP ALUs	2 ALU, 2 Mul, 2 div	2 ALU, 1 Mul, 1 div	1 ALU, 1 Mul, 1 div

(a)



(b)

Figure 5: (a) Multicore system configurations used to evaluate MAMPO (b) Interconnection networks used in the multicore system configurations

for each operand of an instruction is assigned as the destination register of the producer instruction based on the corresponding dependency assignment.

10. The loop counters for the inner loops are set based on the specified working set size and the compare instructions for loop termination are generated by choosing an integer ALU instruction in the code.

11. The second inner loop is also generated, which is a copy of the first loop without the memory operations if the MLP frequency is low or if it is set to be high, the second loop is generated just as a copy of the first loop.

3. EXPERIMENTAL SETUP, RESULTS AND ANALYSIS

To test the efficacy of this power virus generation framework, we use the Virtutech Simics full system simulator along with Wisconsin Multifacet GEMS [24] to evaluate the power consumption of the multithreaded synthetic workloads for the SPARC ISA using Solaris 10 operating system. The cycle accurate out-of-order processor simulator Opal, the detailed memory simulator Ruby and the network simulator Garnet, all of which are a part of GEMS was used to model a typical Chip-MultiProcessor (CMP). The power consumption in the core is evaluated using the power models provided by Wattch [25] for the most aggressive clock gating 'cc3' in Wattch. The power consumption of the shared L2 cache and the directory is modeled with help of the latest power models for caches using CACTI [26].

The power consumption of the network was evaluated using the network power model Orion [27]. The power consumption of DRAM for DDR2 technology was modeled by integrating the DRAMsim simulator [28] into GEMS. The power models used for all the components of the CMP are for a 90nm technology. It is to be noted that this power virus generation methodology aims to help a system designer in the pre-silicon design stage of a system, when only the simulators will be available than real hardware. We have used the GNU gcc compiler for SPARC ISA with the optimization level of O2 for compiling the synthetics and an optimization level of O3 for compiling other workloads.

The Figure 5(a) shows the three multicore system configurations that are used to evaluate the efficacy of MAMPO. Figure 5(b) shows the various interconnection networks used in these multicore systems. We use the most popular MOESI cache coherence protocol for all our experiments, which has the states Modified, Owned, Exclusive, Shared and Invalid for every cache block. We use a multibanked shared L2 cache and a Non-Uniform Memory Access protocol with a directory size of 1 MB. Our power models were validated against published power numbers for the Sun Microsystem's Niagara and the Rock systems by constructing an equivalent system using our infrastructure. For the machine learning, we use IBM's Genetic Algorithm toolset called SNAP [29] [8]. We have used a mutation rate of 0.05, crossover rate of 0.85 and a reproduction rate of 0.10. A population size of 48 individuals per generation was found to be the most optimal deme size for this problem. Increasing it beyond 48 does not help as the execution time of each generation becomes high due to the increased number of chromosomes to evaluate and when the deme size is smaller than 48, the population size is not big enough to search such a large abstract workload space in the same time.

We compare the power consumption of the generated MAMPO virus with that of the power consumption of the PARSEC workloads. In the multithreaded synthetic, we use a feature called MAGIC instruction in Simics to be able to perform detailed simulation for only the core part of the synthetic code. We start the detailed simulation after all the threads have reached the barrier after the initial memory allocation and processor bind system calls. The first thread that reaches the end of it's execution signals Simics to stop the simulation and the profiled data is used to calculate the power consumption using the power models. Typically the number of dy-

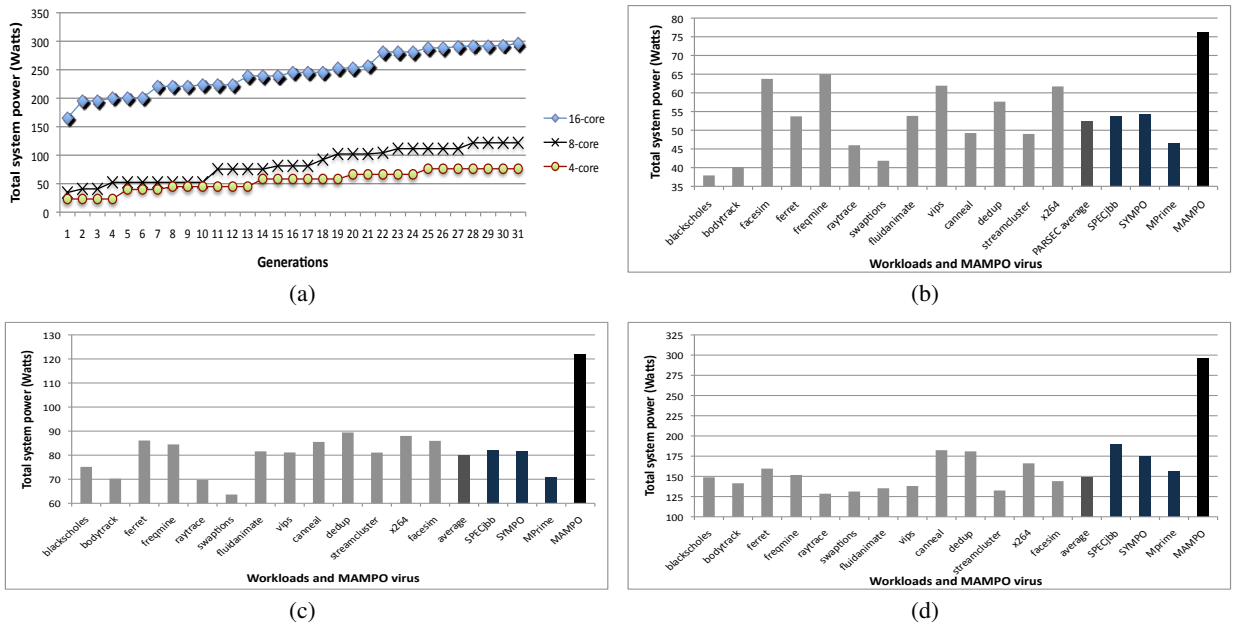


Figure 6: MAMPO virus generation for various system configurations (a) Power consumption of the best power virus at the end of each generation (b) Power for final virus for Machine configuration - I (c) Power for final virus for Machine configuration - II (d) Power for final virus for Machine configuration - III

namic instructions in the multithreaded synthetic is around a few million instructions per thread. For PARSEC workloads, we use the input set provided for detailed microarchitectural simulations called 'simsmall'.

Figure 6(a) shows the power consumption of the best power virus at the end of each generation for approximately 30 generations, after which there is negligible increase in power consumption due to the convergence of the GA. It is to be noted that there are not any known power viruses targeting multicores and so we compare our generated viruses against running multiple copies of single-core power viruses. MPrime [11], which is popularly called the torture test is one of the system-level industry grade power viruses for single-core systems. SYMPO [8] is the most recent previous work by Ganesan et. al to generate a max-power virus for a given single-core system. We have implemented the SYMPO framework to enable us to generate SYMPO viruses for each of our configurations and compare the overall power consumption of running multiple copies of SYMPO viruses, one on each core, with that of MAMPO viruses. Other than these power viruses, we also compare out power viruses with that of the commercial Java benchmark SPECjbb. The number of threads in SPECjbb was set to be equal to the number of processors in the system configuration. Figures 6(b), 6(c) and 6(d) show the comparison of the power consumption of MAMPO viruses with that of the power consumption of the workloads in the PARSEC benchmark suite, MPrime, SYMPO and that of SPECjbb for the three machine configurations as in Figure 5(a). It can be noted that the MAMPO viruses consume 45%, 52% and 98% more power than the average power consumption of the workloads in the PARSEC suite. The MAMPO viruses consume 63%, 72% and 89% more power than that of MPrime and 40%, 49% and 69% more power than that of the SYMPO virus for the three machine configurations respectively, clearly bringing out the importance of such a multithreaded synthetic power virus generation framework compared to running multiple single-core power viruses. The MAMPO viruses consume 41%, 48% and 56% more power than that of the SPECjbb. From these results, it can be observed that the MAMPO virus outperforms the other workloads as the number of cores increases due to the reason that MAMPO is very effective in stressing the interconnection network. It is to be noted that the energy spent in terms of data transfer through the interconnection network is predicted to increase many folds [30] due to global wire scaling problems compared to the energy spent in computation bringing out the significance of their contribution to the power consumption of future systems.

Since the fitness evaluation of the individuals in a generation is independent of each other, they can be run in parallel. Thus, when we use a modern parallel system with many cores, this process of finding a power virus can be done with a good amount of parallelism resulting in a quicker convergence of the GA. The time taken for MAMPO to generate these power viruses for the three system configurations range between 8 to 12 hours on a 3.4 GHz Intel Xeon system with 16 cores. Though we use a full system simulator with cycle accurate models to evaluate the power consumption, the total number of dynamic instructions in the synthetic is restricted to be less than 16 million instructions, to enable this search happen within a reasonable time frame. Rather, to find the same virus manually, a system architect will have to typically spend a few weeks of manpower and can still not be sure if it is a good power virus or not.

The power viruses generated for each of these configurations are found to be having exactly the same number of threads as that of the number of processors. For example, a four-threaded workload is found to be a more suitable candidate for a quad-core system than an eight or sixteen threaded workload. This can be attributed to the fact that the time taken for even a DRAM access in our framework is not enough to force a context switch in the thread scheduler used in Solaris 10. But, a knob like number of threads may be utilized more effectively when a hard disk access is also modeled, where the access latency could force the scheduler to do a context switch. We do not model the components like the chipset and the disk sub-

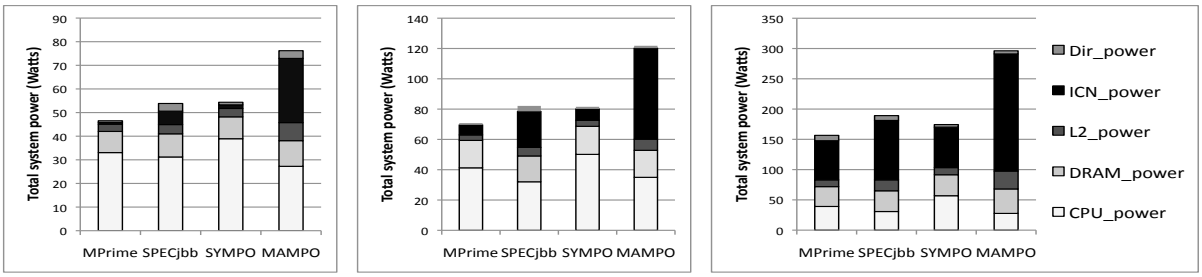


Figure 7: Breakdown of MAMPO virus and comparison to MPrime (a) Machine configuration - I (b) Machine configuration - II (c) Machine configuration - III

system in this study due to the reason that they have nearly constant power consumption over various range of workloads [31].

It would be interesting to see how the characteristics of the finally generated power viruses vary across the different system configurations. Figures 7(a), 7(b) and 7(c) show the breakdown of the power consumption of the MAMPO viruses, SYMPO, MPrime and SPECjbb in various parts of the system. It can be noted that the single-core power viruses SYMPO and MPrime consume maximum power in the cores, rather the MAMPO viruses stress different parts of the system in such a way that the total power is maximized. Some common characteristics of these power viruses are that they have 10-20% of the memory accesses to shared data and they try to move as much data as possible through the interconnection network, besides making sure that the slowdown caused to the CPUs due to this is minimum. The maximum power achieved by our tool is still 'realistically attainable' as the characteristics of the power viruses still map to the range for the abstract workload model parameters of realistic workloads. It is to be noted that the power viruses for each of these systems configurations have different settings for most of the knobs other than the aforementioned ones and it is wasteful to analyze this further due to their sensitivity to the microarchitecture changes and the aim of this whole machine learning based framework is to make this power virus generation a completely automated black-box approach to avoid the need to model the complex interactions involved in the execution of a workload within a multicore system.

4. RELATED WORK

Synthetic benchmark generation: Synthesizing workloads/traces [32] [33] [34] [35] for performance evaluation has always been an area that has been constantly under investigation. Usage of synthetic benchmarks as miniaturized proxies [15] [17] [16] for long running applications has been proven to be a viable solution to use with very slow cycle accurate and Register Transfer Level (RTL) simulators in pre-silicon design validation. Joshi et al [14] [13] introduced the idea of using synthetic benchmarks for cloning proprietary applications that cannot be shared with the processor vendors. Though there has been quite some work targeting the characterization of parallel applications [36], this is the first attempt towards synthesizing multithreaded workloads.

State of the art power viruses: There have been a lot of industry efforts [37] [38] [39] [40] [41] towards hand crafting code snippets to serve as power viruses for the single core processors. Stability testing tools written for overclockers like CPUBurnin [5] and CPUBurn [6] are also popular power viruses. The program MPrime [11], which searches for mersenne prime number is popularly called the torture test and is a well known power virus used in the industry. Joshi et. al [9] and Ganesan et al [8] automated power virus generation for single cores.

5. CONCLUSION

In this paper, we proposed the usage of MAMPO, which is a multithreaded synthetic power virus generation framework targeting multicore processors. We validate the efficacy of MAMPO by comparing the power consumption of the generated virus with that of the workloads in PARSEC for three different multicore system configurations and show that the MAMPO virus consumes 45%, 52% and 98% more power than the average power consumption of the PARSEC workloads. We also show that the single core power viruses, when run on muticore systems do not serve the purpose as a multicore system virus by comparing the power consumption of the MAMPO virus with that of the previously proposed SYMPO viruses and the well known power virus MPrime. The MAMPO virus consumes 40% to 89% more power than running multiple copies of single-core viruses in parallel. We also provide a comparison of the power consumption of the MAMPO virus with that of SPECjbb and show that the MAMPO virus consumes 41%, 48% and 56% more power than that of SPECjbb. Though the power viruses generated by MAMPO cannot theoretically guarantee to be the absolute worst-case, based on the convergence of the Genetic Algorithm run with multiple seeds, we can be sure that the generated power viruses will serve as a tight upper-bound for the maximum power for all practical purposes and such a framework will be a very useful tool for the system designers.

6. ACKNOWLEDGMENTS

This work has been supported and partially funded by SRC under Task ID 1797.001, National Science Foundation under grant numbers 0702694, 0751112, 0750847, 0750851, 0750852, 0750860, 0750868, 0750884 and 0751091, Lockheed Martin, Sun Microsystems and IBM. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or other sponsors.

7. REFERENCES

[1] Stuart Berke, David Moss, and Randy Randall. Understanding the challenges of delivering cost-effective, high- efficiency power supplies. *http://www.dell.com/downloads/global/power/ps2q07-20070270-PowerTCO.pdf*, May 2007.

[2] Xiao Ping Wu, Masataka Mochizuki, Koichi Mashiko, Thang Nguyen, Vijit Wuttijumnong, Gerald Cabsao, and Aliakbar Akbarzadeh Randeep Singh. Energy conservation approach for data center cooling using heat pipe based cold energy storage system. *26th Annual IEEE Semiconductor Thermal Measurement and Management Symposium, 2010. SEMI-THERM 2010 Page(s): 115 - 122*, March 2010.

[3] Michael K Patterson. The Effect of Data Center Temperature on Energy Efficiency. *11th Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems, 2008. ITHERM 2008 Page(s): 1167 - 1174*, May 2008.

[4] Amip Shah, Chandrakant Patel, Cullen Bash, Ratnesh Sharma, and Rocky Shih. Impact of rack-level compaction on the data center cooling ensemble. *11th Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems, 2008. ITHERM 2008 Page(s): 1175 - 1182*, May 2008.

[5] `http://www.softpedia.com/get/System/Benchmarks/CPU-Burnin.shtml`.

[6] `http://pages.sbcglobal.net/redelm`.

[7] Private Communication with Advanced Micro Devices (AMD) Design Engineer.

[8] Karthik Ganesan, Jungho Jo, W. Lloyd Bircher, Dimitris Kaseridis, Zhibin Yu, and Lizy K. John. System-level Max Power (SYMPO) - A systematic approach for escalating system-level power consumption using synthetic benchmarks. *In the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), Vienna, Austria*, September 2010.

[9] Ajay Joshi, Lieven Eeckhout, Lizy K. John, and Ciji Isen. Automated microprocessor stressmark generation. *The 14th International Symposium on High Performance Computer Architecture (HPCA)*, February 2008.

[10] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[11] `http://www.mersenne.org/freesoft`.

[12] L D Davis and Melanie Mitchel. Handbook of genetic algorithms. *Van Nostrand Reinhold*, 1991.

[13] Ajay Joshi, Lieven Eeckhout, Robert H. Bell Jr., and Lizy K. John. Performance Cloning: A Technique for Disseminating Proprietary Applications as Benchmarks. *International Symposium on Workload Characterization*, October 2006.

[14] Ajay Joshi, Lieven Eeckhout, Jr. Robert H. Bell, and Lizy K. John. Distilling the essence of proprietary workloads into miniature benchmarks. *ACM Transactions on Architecture and Code Optimization (TACO 2008)*, August 2008.

[15] Karthik Ganesan, Jungho Jo, and Lizy K John. Synthesizing Memory-Level Parallelism Aware Miniature Clones for SPEC CPU2006 and ImplantBench Workloads. *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2010.

[16] Robert H Bell and Lizy K John. Improved Automatic Test Case Synthesis For Performance Model Validation. *Proceedings of the International Conference on Supercomputing 111-120*, 2005.

[17] Jr Robert H. Bell, Rajiv R. Bhatia, Lizy K. John, Jeff Stuecheli, John Griswell, Paul Tu, Louis Capps, Anton Blanchard, and Ravel Thai. Automatic Testcase Synthesis and Performance Model Validation for High Performance PowerPC Processors. *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2006)*, March 2006.

[18] Nick Barrow-Williams, Christian Fensch, and Simon Moore. A communication Characterization of Splash-2 and Parsec. *IEEE International Symposium on Workload Characterization*, October 2009.

[19] Michael C. Huang Hemayet Hossain, Sandhya Dwarkadas. Improving support for Locality and fine-grain sharing in chip multiprocessors. *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, October 2008.

[20] Liqun Cheng, John B. Carter, and Donglai Dai. An Adaptive Cache Coherence Protocol Optimized for Producer-Consumer Sharing. *IEEE 13th International Symposium on High Performance Computer Architecture, 2007. HPCA 2007*, February 2007.

[21] Umakishore Ramachandran, Gautam Shah, Anand Sivasubramaniam, Aman Singla, and Ivan Yanasak. Architectural Mechanisms for Explicit Communication in Shared Memory Multiprocessors. *Proceedings of the IEEE/ACM Supercomputing Conference*, 1995.

[22] Guhan Viswanathan and James R. Larus. Compiler-directed Shared-Memory Communication for Iterative Parallel Applications. *Proceedings of the ACM/IEEE Conference on Supercomputing*, 1996.

[23] Haungs M, Sallee P, and Farrens M. Branch transition rate: a new metric for improved branchclassification analysis. *Sixth International Symposium on High-Performance Computer Architecture (HPCA 2000), Volume , Issue , 2000 Page(s):241 - 25*, January 2000.

[24] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, , and David A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News (CAN),*, September 2005.

[25] Margaret Martonosi, Vivek Tiwari, and David Brooks. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. *isca, pp.83, 27th Annual International Symposium on Computer Architecture (ISCA 2000)*.

[26] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches With CACTI 6.0. *Proc. 40th Annual IEEE/ACM IntâĂŹl Symp. on Microarchitecture (MICRO 07), IEEE CS Press pp. 3-14.*, December 2007.

[27] Hangsheng Wang, Xinping Zhu, Li-Shiuan Peh, and Sharad Malik. Orion: A Power-Performance Simulator for Interconnection Networks. *In Proceedings of MICRO 35, Istanbul, Turkey*, November 2002.

[28] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Katie Baynes, Aamer Jaleel, and Bruce Jacob. DRAMsim: A memory-system simulator. *Computer Arch. News, vol. 33, no. 4, pp. 100-107*, Sep 2005.

[29] Sameh Sharkawi, Don Desota, Raj P, Rajeev Indukuru, Stephen Stevens, and Valerie Taylor. Performance Projection of HPC Applications Using SPEC CFP2006 Benchmarks. *IEEE International Parallel & Distributed Processing Symp.*, May 2009.

[30] Michele Petracca, Benjamin G. Lee, Keren Bergman, and

Luca P. Carloni. Design Exploration of Optical Interconnection Networks for Chip Multiprocessors. *16th IEEE Symposium on High Performance Interconnects pages: 31 - 40*, September 2008.

[31] W. Lloyd Bircher and Lizy K.John. Complete System Power Estimation: A Trickle-Down Approach Based on Performance Events. *International Symposium on Performance Analysis of Systems and Software*, April 2007.

[32] Cheng-Ta Hsieh and M. Pedram. Microprocessor power estimation using profile-driven program synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,*, November 1998.

[33] Wing Shing Wong and Robert J. T. Morris. Benchmark Synthesis Using the LRU Cache Hit Function. *IEEE Transactions on Computers*, 1988.

[34] E.S. Sorenson and J.K. Flanagan. Evaluating synthetic trace models using locality surfaces. *2002. WWC-5. 2002 IEEE International Workshop on Workload Characterization*, November 2002.

[35] Lizy John, Jungho Jo, and Karthik Ganesan. Workload Synthesis for a Communications SoC. *In Workshop on SoC Architecture, Accelerators and Workloads, held in conjunction with HPCA-17, San Antonio, Texas, Feb 2011*.

[36] Karthik Ganesan, Lizy K John, James Sexton, and Valentina Salapura. A Performance Counter Based Workload Characterization on BlueGene/P. *In 37th International Conference on Parallel Processing (ICPP), Portland, Oregon, September 2008*.

[37] W. Felter and T. Keller. Power measurement on the apple power mac g5. *IBM Tech Report RC23276*, 2004.

[38] M. Gowan, L. Biro, and D. Jackson. Power considerations in the design of the alpha 21264 microprocessor. *Design Automation Conference*, 1998.

[39] R. Vishwanath, V. Wakharkar, A. Watwe, and V.Lebonheur. Thermal performance challenges from silicon to systems. *Intel Technology Journal*, 2000.

[40] R. Joseph, D. Brooks, and M. Martonosi. Control techniques to eliminate voltage emergencies in high performance processors. *High Performance Computer Architectures*, 2003.

[41] F. Najm, S. Goel, and I. Hajj. Power estimation in sequential circuits. *Design Automation Conference*, 1995.