

Automated Microprocessor Stressmark Generation

Ajay M. Joshi* Lieven Eeckhout** Lizy K. John* Ciji Isen*

**The University of Texas at Austin*

***Ghent University, Belgium*

ajoshi@ece.utexas.edu, leeckhou@elis.ugent.be, ljohn@ece.utexas.edu

Abstract

Estimating the maximum power and thermal characteristics of a processor is essential for designing its power delivery system, packaging, cooling, and power/thermal management schemes. Typical benchmark suites used in performance evaluation do not stress the processor to its limit though, and current practice in industry is to develop artificial benchmarks that are specifically written to generate maximum processor (component) activity. However, manually developing and tuning so called stressmarks is extremely tedious and time-consuming while requiring an intimate understanding of the processor.

A synthetic program that can be tuned to produce a variety of benchmark characteristics would significantly help in addressing this problem by enabling the automatic exploration of the large temperature and power design space. This paper demonstrates that with a suitable choice of only 40 hardware-independent program characteristics related to the instruction mix, instruction-level parallelism, control flow behavior, and memory access patterns, it is possible to generate a synthetic benchmark whose performance relates to that of general-purpose and commercial applications. Leveraging this abstract workload modeling approach, we propose StressMaker, a framework that uses machine learning for the automated generation of stressmarks. A comparison with an exhaustive exploration of a large power design space demonstrates that StressMaker is very effective in automatically generating stressmarks in a limited amount of time.

1. Introduction

In recent years, energy, power, power density, thermal hot spots, voltage variation, etc., have emerged as first-class constraints in the design of high-performance microprocessors [5][12][13][14][18][30]. As a result, along with performance, it has become extremely important to measure and analyze power, energy, and temperature related design concerns at all stages in a microprocessor design flow – from early-

stage exploration, microarchitecture definition, register-transfer-level (RTL) description, to circuit-level implementation.

In order to design a power- and temperature-aware microprocessor it is not only important to evaluate the design's power, energy and thermal characteristics when executing a typical workload, but also to evaluate its *maximum* power and operating temperature characteristics. In other words, it is also important to analyze the impact of application code sequences that could stress the processor's power and thermal characteristics – although these code sequences are infrequent and may only occur in a short burst [13][28][34]. Worst-case maximum power dissipation and operating temperature characterization is essential for evaluating dynamic power and temperature management strategies. Also, large instantaneous and localized power dissipation can cause overheating (hotspots) that can reduce the lifetime of a chip, degrade circuit performance, introduce timing errors, or even result in chip failure [30]. Estimating the maximum power dissipation and operating temperature of a processor is also vital for designing the thermal package (heat sink, cooling, etc.) for the chip and the power supply for the system [34]. As such, characterizing the maximum thermal characteristics and power limits is greatly needed by microarchitects, circuit designers, and electrical engineers.

Industry-standard benchmarks though do not stress a processor to its limit, and are not particularly useful when characterizing the maximum power and thermal requirements of a design. Benchmarking committees such as the Standard Performance Evaluation Consortium (SPEC) and the EDN Embedded Microprocessor Benchmark Consortium (EEMBC) have recognized the need for power and energy oriented benchmarks, and are in the process of developing such benchmark suites [21][32]. However, these benchmarks too will only represent typical power consumption and not the worst-case maximum power dissipation. Due to the lack of any standardized stress benchmarks, current practice in industry is to develop hand-coded synthetic 'max-power' benchmarks, or *stressmarks*, that are specifically written to generate

maximum power consumption for a particular processor [2][13][28][34].

Developing stressmarks is both time-consuming and tedious. For example, a max-power stressmark has to generate maximum and simultaneous activity in each of the processor components; similarly, a thermal stressmark not only has to deal with power consumption but also with lateral coupling among microarchitecture blocks, role of the heat sink, etc. [30]. This requires a very detailed knowledge of the processor design [13], and given the complexity of modern day high-performance superscalar microprocessors, writing and tuning a stressmark can take up to several weeks [2]. In addition, given that a stressmark is tied to a specific processor, exploring multiple processor architectures in terms of their maximum power consumption and/or thermal characteristics quickly becomes infeasible and may stretch the time-to-market.

In this paper we propose *StressMaker*, a framework for the automated generation of stressmarks. The key enabler to *StressMaker* is the ability to generate a synthetic benchmark from an abstract workload model. *StressMaker* explores the workload space by ‘turning knobs’ in the abstract workload model, and uses machine learning for driving the search for stressmarks.

In this paper, we make three major contributions:

- We identify a limited set of hardware-independent program characteristics that collectively represent the abstract workload model. The key program characteristics relate to the instruction mix, instruction-level parallelism, control flow behavior, and memory access patterns. When used to generate a synthetic benchmark, the abstract workload model represents real-world workload behavior. Our experimental results using the SPEC CPU2000 benchmarks and three commercial workloads (SPECjbb2005, DBT2, and DBMS) report an average performance and power deviation of 10% and 7%, respectively, when comparing a real workload against its synthetic clone.
- We propose *StressMaker*, a novel approach to automatically generate synthetic stressmarks for microprocessor design studies. *StressMaker* uses machine learning to explore the workload space by varying the program characteristics in the abstract workload model in search for stressmarks. The important advantage of *StressMaker*, next to being fully automated, is that it enables generating stressmarks for cases where manually writing a stressmark is difficult because of the complex hardware/software interactions in today’s high-performance microprocessors. We demonstrate the

feasibility and value of *StressMaker* by designing max-power, max-temperature, and dI/dt stressmarks. These stressmarks stress the processor much more than typical workloads, are close to optimal compared to an exhaustive search of the workload behavior space, and could serve as a starting point for detailed power/thermal analysis.

- We develop a framework, *BenchMaker*, which is parameterized to generate synthetic benchmarks that can be executed on real hardware, execution-driven simulators, and RTL models. *StressMaker* is just one of the many useful applications of *BenchMaker*. The parameterized nature of *BenchMaker* makes it an invaluable tool for exploring the workload space, and for gaining insight into how performance is affected by high-level program characteristics. The computer architecture research community has recognized the need for developing parameterized workloads [31], and we believe that *BenchMaker* is a significant step towards achieving that goal.

2. *BenchMaker*: Generating Parameterized Synthetic Workloads

The key enabler to *StressMaker* is the ability to describe a synthetic benchmark from an abstract workload model. Figure 1 illustrates *BenchMaker*, our framework for generating synthetic benchmarks from a set of hardware-independent program characteristics. The program characteristics measure the inherent behavioral properties of the program, and, collectively, form the abstract workload model. This abstract workload model serves as input to the synthetic benchmark generator. The goal of parameterized synthetic workload modeling is to maintain good representativeness and good accuracy with a *limited* number of workload characteristics. To do so, we capture them at a coarse granularity using average statistics over the entire program. This is in contrast to prior work on synthetic benchmark generation [1][20] which models program characteristics at a fine granularity by capturing characteristics at the basic block and/or path level. Although measuring characteristics at a coarse granularity likely reduces the representativeness of the synthetic benchmarks compared to fine grained characteristics, this is key to enable the flexibility for generating benchmarks with characteristics of interest by simply ‘turning’ workload behavior ‘knobs’ – this property will be exploited in *StressMaker* for the automatic generation of stressmarks. In the next two subsections we describe the workload characteristics and the synthesis algorithm.

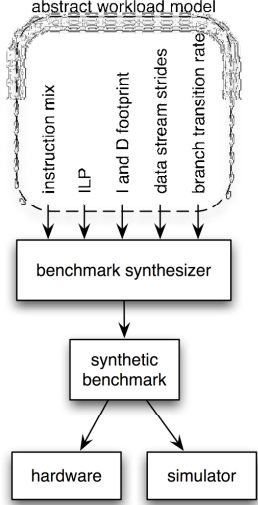


Figure 1: BenchMaker framework for constructing parameterized synthetic benchmarks.

2.1. Abstract Workload Model

We use a collection of fundamental program characteristics that are independent of the underlying microarchitecture, and which relate to the instruction mix, control flow predictability, instruction-level-parallelism (ILP), data locality, and instruction locality. We now describe these workload characteristics.

Instruction Mix. The instruction mix of a program captures the relative frequency of short-latency and long-latency integer and floating-point operations, loads, stores and branches occurring in the dynamic instruction stream of a program.

Instruction-Level Parallelism (ILP). The ILP is modeled by means of the inter-operation dependency distance, defined as the number of instructions in the dynamic instruction stream between the production (write) and consumption (read) of a register and/or memory value. We describe the inter-operation dependency distance as a cumulative distribution organized in eight buckets.

Instruction and Data Footprint. We measure the instruction and data footprint of a program in terms of the total number of unique instruction and data addresses referenced by the program.

Data Stream Strides. We model the data stream by means of a distribution of the local data strides, which was shown to be accurate for a variety of workloads, including pointer-chasing codes, see [19]. A local stride is defined as the delta in data memory addresses between successive memory accesses by a single static instruction. We describe the local strides in terms of 32-byte block sizes (analogous to a cache line size), i.e., stride 0 refers to a local data stride of 0 to 31 bytes (consecutive addresses are within one cache line distance). The local strides are summarized as a

histogram showing the percentage of memory access instructions with stride values of 0, 1, 2, etc.

Branch Transition Rate. In order to model varying levels of control flow predictability we use an attribute called the branch transition rate [15]. The transition rate of a static branch is defined as the number of times it switches between taken and not-taken directions as it is executed, divided by the total number of times that the branch is executed. By definition, branches with very low transition rates are always biased towards either taken or not-taken, and are easy to predict. However, branches that transition between taken and not-taken sequences at a moderate rate are relatively more difficult to predict.

To summarize, the 40 workload characteristics constituting the abstract workload model are described in Table 1. These workloads characteristics cover a wide range of program characteristics that affect overall workload behavior. If needed, the abstract workload model can be enhanced to model additional characteristics such as operand data values, hamming distances between consecutive instruction opcodes, etc.

Table 1. Microarchitecture-independent characteristics constituting an abstract workload model.

Category	No.	Characteristic
insn mix	8	fraction integer short-latency insns
		fraction integer long-latency insns
		fraction fp short-latency insns
		fraction fp long-latency insns
		fraction integer loads
		fraction integer stores
		fraction fp loads
		fraction fp stores
ILP	8	8 probabilities constituting the register dependency distance distribution: dependency distance equal to 1 insn (insn is dependent on the previous insn in the dynamic insn stream), smaller than 2, 4, 6, 8, 16, 32, and greater than 32 insns
data locality	1	data footprint
	10	distribution of local stride values organized in 10 buckets
insn locality	1	instruction footprint
predictability	10	distribution of branch transition rate organized in 10 buckets
	2	avg and stdev of the dynamic basic block size

2.2. Workload Synthesis Algorithm

We now describe the algorithm to synthesize a synthetic benchmark from the abstract workload model – this algorithm is based on our prior work [20]. Benchmark synthesis comprises of five sub steps: (i) generating the synthetic benchmark spine using instruction mix and basic block analysis, (ii) memory access modeling, (iii) modeling branch predictability, (iv) register assignment, and (v) code generation.

2.2.1. Generating Program Spine. A normal distribution function based on the average basic block size and its standard deviation is used to generate a linear chain of basic blocks. This linear chain of basic blocks forms the spine of the synthetic benchmark program. We use the instruction footprint of the program to decide on the length of the spine. After the spine has been instantiated, each basic block is populated based on the instruction mix characteristics, and each instruction operand is assigned a dependency distance – this is done using random number generation on the cumulative dependency distance distribution.

2.2.2. Modeling Memory Access Patterns. For each memory access instruction in the synthetic benchmark we assign a stride value from the stride distribution function. The load or store instruction’s memory access patterns are modeled as a bounded stream of circular references, i.e., each memory operation walks through an array using the stride value assigned to it and then restarts from the first element of the array. The length of each array is simply the ratio of the data footprint of the program and the total number of static load or store instructions in the program.

2.2.3. Modeling Branch Predictability. For each static branch in the spine of the program we assign a transition rate based on the specified transition rate distribution. We achieve this by configuring each basic block in the synthetic stream of instructions to alternate between taken and not-taken directions, such that the branch exhibits the desired transition rate at run time. A counter is incremented on each iteration count, and a modulo operation is used to decide whether the branch is taken or not taken.

2.2.4. Register Assignment. In this step we use the dependency distances that were assigned to each instruction to assign register names. The number of registers that are used to satisfy the dependency distances is typically kept to a small value (typically around 10) to prevent the compiler from generating spill code.

2.2.5. Code Generation. During the code generation phase, the instructions are emitted out with a header written in C, which contains initialization code that allocates memory using the *malloc* library call for modeling the memory access patterns. Each instruction is then emitted out with assembly code using *asm* statements embedded in C code. The instructions are targeted towards a specific ISA, Alpha in our case. However, the code generator can be modified to emit instructions for an ISA of interest. The *volatile*

directive is used to prevent the compiler from reordering the sequence of instructions and changing the program characteristics in the synthetic benchmark. The entire program spine is executed in a loop whose number of iterations can be configured to control the dynamic instruction count of the program. This value is tuned to ensure that the synthetic benchmark’s execution characteristics converge to a stable value. Based on our experiments, for the workload characteristics used in this study, the synthetic benchmark execution converges to steady state in a maximum of 10 million dynamic instructions.

3. StressMaker: Building Stressmarks

The flow chart in Figure 2 illustrates the approach used by StressMaker to automatically generate stressmarks. In the first step, a synthetic benchmark is synthesized using BenchMaker from randomly chosen values for all the program characteristics in the abstract workload model. In the second step, the synthetic benchmark is run on the microprocessor model, and the value of the optimization objective to be stressed, such as power or temperature, is measured. The model used for simulation can be a high-level performance/power model, an RTL-level Verilog model, or a circuit-level implementation. In the third step, a decision to continue or stop is made based on the stress level placed by the synthetic benchmark. In the fourth step, machine learning is used to alter the values of the program characteristics in order to improve the stress level of the corresponding synthetic benchmark. This iterative process continues until the search process converges. The end result is a stressmark, a synthetic benchmark that optimizes the stress criterion of interest.

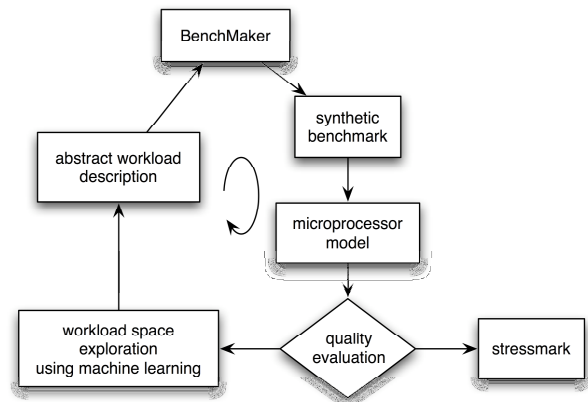


Figure 2: StressMaker framework.

The workload space built up by the abstract workload model is extremely large, and by consequence it is impossible to evaluate every design

point. Therefore, we use a genetic algorithm to automatically search and prune the workload space to converge on a set of workload attributes that maximize an objective function of interest, such as power, temperature, etc. The goal of the genetic algorithm is to intelligently search the workload space by varying the workload characteristics in the abstract workload description, and optimize those characteristics towards a stressmark. *Genetic Search* initially randomly selects a set of design points, called a *generation*: these design points are randomly chosen abstract workload configurations. These design points are subsequently evaluated according to the objective function, also called the *fitness function*, e.g., maximum average power, maximum temperature, etc. – evaluating the fitness function of a design point requires simulating the corresponding synthetic benchmark. A new population, an *offspring*, which is a subset of these design points, is probabilistically selected by weighting the design points’ *fitness* functions, i.e., a fitter design point is more likely to be selected. Selection alone cannot introduce new design points in the search space, therefore *mutation* and *crossover* are performed to build the *offspring* generation. *Crossover* is performed, with probability p_{cross} , by randomly exchanging parts of two selected design points from the current generation. The *mutation* operator prevents premature convergence to local optima by randomly altering parts of a design point, with a small probability p_{mut} . The generational process is continued until a specified termination condition has been reached. In our experiments we specify the termination condition as the point when there is little or no improvement in the objective function across successive generations. We use the genetic search algorithm with p_{cross} and p_{mut} set to 0.95 and 0.02, respectively. The end result of the genetic algorithm is an abstract workload configuration of which its synthetic benchmark stresses the objective function the most – this is the stressmark.

4. Experimental Setup

4.1. Simulation Infrastructure

For evaluating BenchMaker, we use the `sim-alpha` simulator that has been validated against the superscalar out-of-order Alpha 21264 processor [9]. For our StressMaker experiments we use the `sim-outorder` simulator from the SimpleScalar Toolset v3.0. In order to estimate the power characteristics of the benchmarks we use an architectural power modeling tool, namely Wattch v1.02 [5] which was shown to provide good relative accuracy, and consider an aggressive clock gating mechanism (`cc3`). We use the `hotfloorplanner` tool to develop a layout for

the `sim-outorder` pipeline, and use the HotSpot v3.1 tool to estimate the steady-state operating temperature based on average power [30]. The stressmarks are compiled using `gcc`, and are simulated for 10 million dynamic instructions. This small dynamic instruction count serves the needs in this paper, however, in case longer-running applications need to be considered, e.g., when studying the effect of temperature on (leakage) power consumption, the stressmarks can also be executed in a loop for a longer time. It should also be noted that StressMaker is agnostic to the underlying simulation model, and can be easily ported to a more accurate industry-standard simulators and/or power/temperature models.

4.2. Benchmarks

In order to evaluate the parameterized workload synthesis framework, we consider the SPEC CPU2000 benchmarks and select one representative 100M-instruction simulation point selected using SimPoint [29]. We also use traces from three commercial workloads – SPECjbb2005 (Java server workload), DBT2 (OLTP workload), and DBMS (a database management system workload). The commercial workload traces represent 30 million instructions once steady-state has been reached (all warehouses have been loaded), and were generated using the Simics full-system simulator.

4.3. Stressmark Design Space

The workload characteristics form a multi-dimensional space (instruction mix, ILP, branch predictability, instruction footprint, data footprint, and data stream strides). We bound the stressmark search space by discretizing and restricting the values along each dimension, see Table 2. This discretization does not affect the generality of the proposed methodology though – its purpose is to keep the evaluation in this paper tractable. The total design space comprises of 250K points. We will evaluate the efficacy of the genetic search algorithm used in StressMaker against an exhaustive search in this 250K-points search space.

4.4. Microarchitecture Configurations

Table 3 summarizes the three different microarchitecture configurations considered in this paper, ranging from a modest 2-way configuration representative of an embedded microprocessor, to a very aggressive 8-way issue high-performance microprocessor. We use Config 2, a 4-wide superscalar processor, as the base configuration for our experiments.

Table 2. Stressmark design space.

Dimension	No.	Values/Ranges
instruction mix and basic block size	10	Combinations where int, fp, load, store, and branch insns are set to low (10%), moderate (40%), and high (80%)
instruction-level-parallelism	10	Varying from all instructions with virtually no dependencies (dependency distance > 32 instructions) to all instructions are dependent on the prior instruction (dependency distance of 1)
data footprint	5	50K, 100K, 500K, 2M, and 5M unique data addresses
data stream stride distribution	10	Varying from 100% references with stride 0, up to 10% with stride 0 and 90% with stride 10.
instruction footprint	5	600, 1800, 6000, and 20,000 unique instructions
branch predictability	10	Varying from 100% branches with transition rate below 10% to equal distribution of transition rate across all 10 transition rate categories (0-10%, 10-20%, etc.)

Table 3. Microarchitecture configurations evaluated.

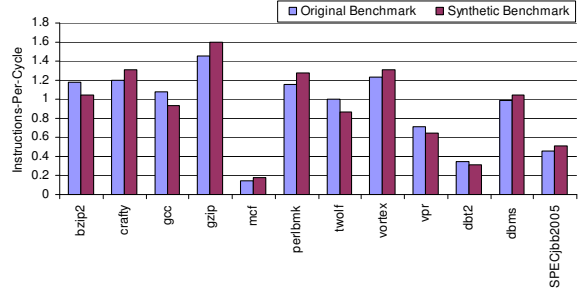
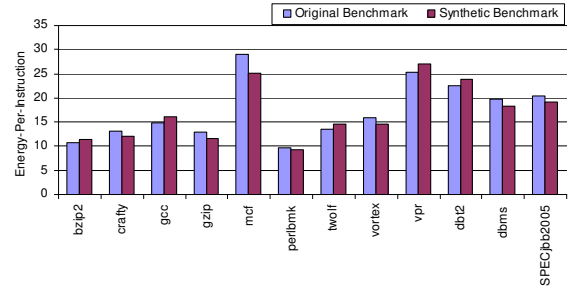
	Config 1	Config 2	Config 3
<i>L1 I- and D-cache</i>	16 KB 2-way	32 KB 4-way	64 KB 4-way
<i>Processor width</i>	2-wide	4-wide	8-wide
<i>Branch predictor</i>	2-level	hybrid 4KB	hybrid 4KB
<i>L2 cache</i>	256KB 4-way	4MB 8-way	4MB 8-way
<i>ROB / LSQ</i>	16 / 8	128 / 64	256 / 128
<i>Functional units</i>	2 int 1 fp	4 int 2 fp	8 int 4fp
<i>MEM access time</i>	40 cycles	150 cycles	150 cycles

5. Evaluation of BenchMaker

In this section we evaluate BenchMaker’s accuracy by using it to generate synthetic benchmark versions of general-purpose (SPEC CPU2000 integer) and commercial (SPECjbb2005, DBT2, and DBMS) workloads – we obtain similar results for the SPEC CPU2000 floating-point benchmarks, and refer to [19] for a detailed analysis. We measure the program characteristics of the SPEC CPU2000 and commercial workloads and feed this abstract workload model to the BenchMaker framework to generate a synthetic clone benchmark with a 10M dynamic instruction count; we then compare the performance/power characteristics of the synthetic benchmark against the original workload.

Figure 3 evaluates the accuracy of BenchMaker for estimating the pipeline instruction throughput

measured in Instructions-Per-Cycle (IPC). We observe that the synthetic benchmark performance numbers track the real benchmark performance numbers very well. The average IPC prediction error is 10.9% and the maximum error is observed for mcf (19.9%). Figure 4 shows similar results for the Energy-Per-Instruction (EPI) metric. The average error in estimating EPI from the synthetic benchmark is 7.5%, with a maximum error of 13.1% for mcf.

**Figure 3: IPC for original and synthetic benchmarks.****Figure 4: EPI for original and synthetic benchmarks.**

Parameterization of workload metrics makes it possible to succinctly describe an application’s behavior using an abstract model with only a limited number (40) of fundamental coarse-grain program characteristics. This is in contrast to prior work in synthetic benchmark generation, which requires several thousands of fine-grain program characteristics [1][20]. BenchMaker trades accuracy (10.9% average error in IPC compared to less than 6% error in our prior work [20]) for flexibility to enable one to easily alter program characteristics and workload behavior.

6. Evaluation of StressMaker

We now evaluate StressMaker by generating various flavors of power and thermal stressmarks. Specifically, we apply StressMaker to automatically construct stressmarks for characterizing maximum average and single-cycle power, dI/dt stressmarks, thermal hotspots, and thermal stress patterns. We also evaluate the efficacy of StressMaker by comparing it

against an exhaustive search of 250K points in the power design space.

6.1. Maximum Sustainable Power

The maximum sustainable power is the maximum average power that can be sustained indefinitely over many clock cycles. Estimating the maximum sustainable power is important for the design of the power delivery system and also the packaging requirements for the microprocessor. We use StressMaker to construct a stressmark for characterizing the maximum sustainable power of the baseline 4-wide microarchitecture (Config 2).

Figure 5 shows a plot of the value of the best *fitness function* (maximum power consumption) in each generation during the iterative process of stress benchmark synthesis using the genetic algorithm. We terminate the search after 15 generations, requiring a total of 225 simulations. The number of generations required before the fitness function can be accepted depends on the search space and the microarchitecture. However, our experiments on three very different microarchitectures suggest that there is little improvement beyond 15 generations, and therefore for our experiments we terminate the search after 15 generations.

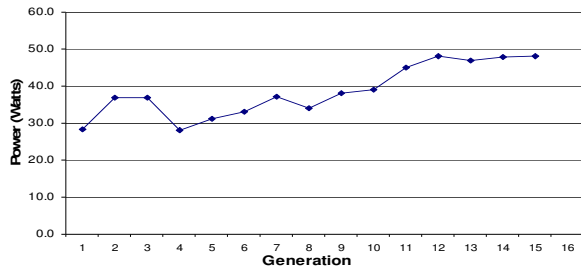


Figure 5: Convergence of StressMaker: the maximum average power consumption for the stressmark across the multiple generations of the genetic search algorithm.

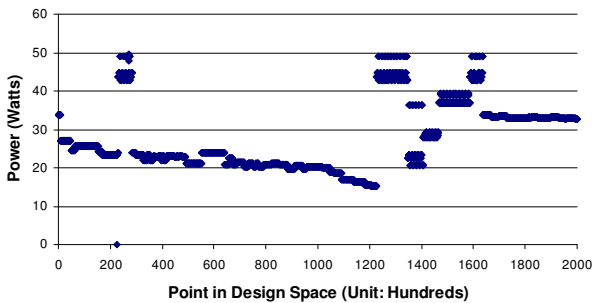


Figure 6: Power consumption for all 250K points in the workload design built up from Table 2.

This ‘maximum sustainable power’ search process results in a stressmark that has a maximum average sustainable power-per-cycle of 48.8W. Figure 6 shows the results of an exhaustive search across all the 250K design points. These results show that the power of the stressmark is within 1% of the maximum power from the exhaustive search, i.e., the stressmark obtained through genetic searching achieves 99% of the maximum power observed from an exhaustive stressmark enumeration. In other words, StressMaker is highly effective in finding a stressmark, and also results in a three orders of magnitude speedup compared to exhaustive searching (225 versus 250K simulations). Automatically generating the stressmark on a 2GHz Intel Pentium Xeon processor using a cross compiler for Alpha and the sim-outorder performance model, takes 2.5 hours. Therefore, we believe StressMaker is an invaluable approach for an expert, because it can quickly narrow down a design space, and provide a stressmark that can be hand tuned to exercise worst-case behavior.

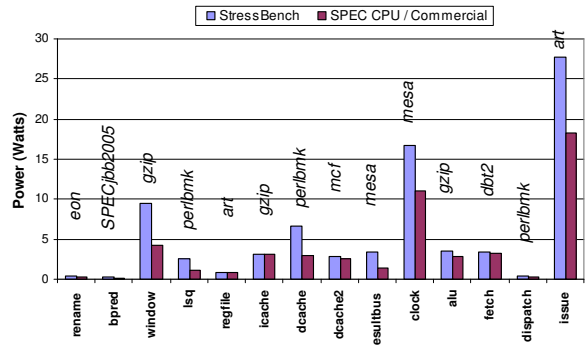


Figure 7: Comparison of the power dissipation in the various microarchitecture units using stressmarks versus the max power consumption observed across all SPEC CPU2000 and commercial benchmarks.

Figure 7 shows the maximum power dissipation of different microarchitecture units using the stressmark, along with the maximum power dissipation of that unit across all SPEC CPU2000 and commercial benchmarks – the benchmark labels in Figure 7 state which benchmark achieves the highest max power per microarchitecture unit, e.g., art achieves the highest power consumption (18W) in the issue logic across all benchmarks whereas the stressmark consumes 28W. The stressmark exercises all the microarchitecture units more than any of these benchmarks. In particular, the stressmark causes significantly higher power dissipation in the instruction window, L1 data cache, clock tree, and the issue logic.

The workload characteristics of the max power stressmark are: (1) Instruction mix of 40% short-latency floating-point operations, 40% short-latency

integer operations, 10% branch instructions, and 10% memory operations; (2) Mostly register dependency distances of greater than 32 instructions, i.e., very high level of ILP, however there still are some dependencies to fill up the issue queue; (3) 80% of branches having a transition rate of less than 10%, and the remaining 20% branches have a transition rate between 10-20% – recall that branches with very low transition rates are highly predictable; (4) Data strides having 95% of the references to the same cache line and 5% with references to the next cache line; (5) Instruction footprint of 1800 instructions; and (6) Data Footprint of 100K bytes. These workload characteristics suggest that the stressmark creates a scenario where the control flow of the program is highly predictable and hence there are no pipeline flushes, the functional units are kept busy, the issue logic does not stall due to large dependency distances, and the locality of the program is such that the data and instruction cache hit rates are extremely high. The characteristics of this stress benchmark are similar to the hand-crafted tests [2][13] that are tuned to maximize processor activity by fully and continuously utilizing the instruction issue logic, all of the execution units, and the major buses. However, the advantage over current practice in building hand-coded max-power stressmarks is that StressMaker provides an automatic process, resulting in substantial savings in time and effort. Also, the automated search process through a large workload space increases confidence in the results.

6.2. Maximum Single-Cycle Power

Maximum single-cycle power is defined as the maximum total power consumed during one clock cycle, and is important to estimate the maximum instantaneous current that can be drawn by the microprocessor from the power supply. We apply the StressMaker framework to automatically construct a stressmark that maximizes single-cycle power. The search process results in a benchmark that has a maximum single-cycle power dissipation of 72W. The workload characteristics of this benchmark are: (1) Instruction mix of 40% long-latency operations, 20% branches, and 40% memory operations; (2) Register dependency distance of greater than 32 instructions, i.e., very high level of ILP; (3) Equal distribution of branch transition rate across all the 10 categories; (4) 10% of the data references have a local stride of 0, 10% a stride of 1, and 80% have a stride of 3 cache lines; (5) Instruction footprint of 1800 instructions; and (6) Data footprint of 5M unique address. These characteristics suggest that the stressmark does not yield the best possible performance due to a mix of easy and difficult to predict branches (evenly distributed transition rates), possible issue stalls (large

percentage of long-latency operations), and data cache misses (large footprint and strides). Therefore, it is not surprising that the average power consumption of this stressmark is only 32W. However, the overlapping of various events creates a condition where all units are simultaneously busy within a single cycle.

Interestingly, the stressmark that maximizes the average sustainable power (Section 6.1) only has a maximum single-cycle power of 59.5W, and cannot be used to estimate maximum single-cycle power. Also, the maximum single-cycle power requirement of a SPEC CPU benchmark, mgrid, is only 57W. This demonstrates that the sequence of instructions resulting in maximum single-cycle power is very timing sensitive – even benchmarks that run for billions of cycles may not probabilistically hit upon this condition.

To further validate StressMaker, the maximum instantaneous power consumption assuming all units are 100% active is 85W – this was computed by summing the power consumption of all the individual microarchitecture units and reflects the theoretical maximum. The 72W attained by the single-cycle max-power stressmark achieves almost 85% of this maximum theoretical power consumption.

6.3. dI/dt Stressmarks

The dI/dt problem refers to large current swings leading to ripples on the supply voltage lines that may cause circuits to fail; the power delivery system should be able to handle these current swings. A dI/dt stressmark, which alternates high and low power consumption over short periods of time, can be used to characterize a microprocessor's susceptibility to the dI/dt problem. Joseph et al. [18] have expressed the need for constructing dI/dt stressmarks and state that manually developing such a stressmark is extremely difficult due to knowledge required about the power, packaging, and timing characteristics of the processor. In order to study the applicability of StressMaker to automatically construct a dI/dt stressmark, we use StressMaker to generate two sequences of 200 instructions – one for maximum single-cycle power and the other for minimum single-cycle power. We then concatenate these two sequences of instructions and evaluate its power characteristics. Our experiments show that the power consumption in the benchmark shows a cyclic behavior at a period of 400 instructions – with 72W and 16W as the maximum and minimum single-cycle power consumption, respectively. Also, it is possible to change the frequency of the power oscillations by varying the number of instructions of the individual (maximum and single-cycle power) stressmarks. These experiments show that it is indeed possible to automatically generate a dI/dt stressmark, which is typically very difficult to hand-craft and tune.

6.4. Comparing Stressmarks Across Microarchitectures

We now study the sensitivity of stressmarks to a specific processor. We therefore generate max-power stressmarks for the three different microarchitectures described in Table 3, and analyze whether the stressmarks are similar across microarchitectures. The stressmarks generated for the three microarchitecture configurations are called stressmarks 1, 2 and 3, respectively. We then execute the three stressmarks on all three microarchitecture configurations. Figure 8 shows the power consumption for each of the stressmarks on all three configurations. We observe that the stressmark synthesized for each microarchitecture configuration always results in maximum power consumption compared to the other two stressmarks, i.e., a stress benchmark generated for one microarchitecture does not result in maximum power for a different microarchitecture. In fact, a stressmark developed for one microarchitecture can result in extremely low power consumption on another microarchitecture, see for example Stressmark1 on Config3.

The three stress benchmarks are similar in that they have highly predictable branches, small instruction and data footprints, and very large register dependency distances. However, their instruction mixes of computational operations are very different, depending on the number of functional units available per type.

We conclude that the characteristics of stressmarks vary across microarchitecture designs. Therefore, separate custom stressmarks have to be constructed for different microarchitectures. This further motivates the importance of having an automated framework for generating stressmarks.

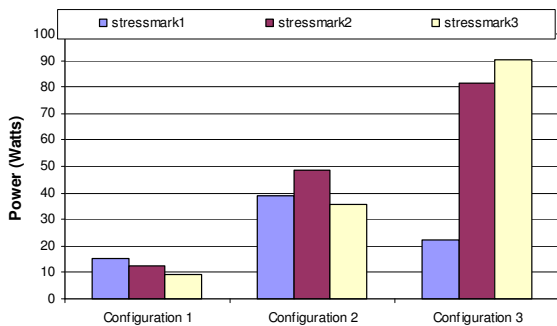


Figure 8: Mutual comparison of stressmarks and microarchitectures.

6.5. Creating Thermal Hotspots

Applications can cause localized heating of specific units of a microarchitecture design, called hotspots, which can cause timing errors, and/or permanent chip damage. Therefore, to study the impact of hotspots in different microarchitecture units it is important to design stressmarks that can be used to vary the location of a hotspot [30].

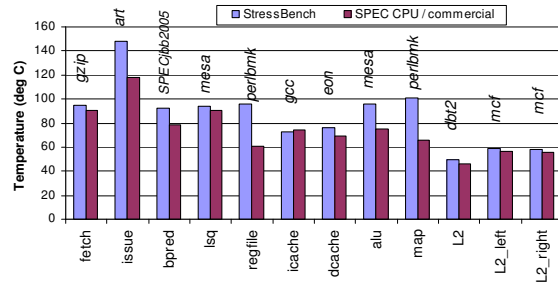


Figure 9: Comparison of the hotspots created by the stressmarks versus the SPEC CPU2000 and commercial benchmarks.

We apply StressMaker to generate stressmarks that can create hotspots across different microarchitecture units on the floorplan. Figure 9 compares hotspots generated by StressMaker with the hotspots generated by the SPEC CPU2000 and commercial benchmarks. As compared to these benchmarks, the stressmarks are very effective in creating hotspots in the issue, register file, execution, and register remap units.

6.6. Thermal Stress Patterns

In order to support dynamic thermal management schemes it has become important to place on-chip sensors to monitor temperature at different locations on the chip. Conceptually, there can be applications that only stress a particular unit that is far from a sensor, causing hotspots that may not be visible to the distant sensor causing permanent damage to the chip [14][23]. Typically, only a few sensors can be placed on a chip. Therefore, the placement of sensors needs to be optimized based on the maximum thermal differential that can exist between different units on the chip. Hand-crafted tests have been typically used to develop such differentials [23]. StressMaker seems to be a natural way to optimize a complex objective function such as the temperature differential between two microarchitecture units. We selected a set of microarchitecture units and generated stressmarks to maximize the temperature difference between units that are not adjacent to each other. Table 4 shows the pair of units, maximum temperature differential created by

the automatically generated stressmark, and the stressmark’s key behavioral characteristics.

Table 4. Developing thermal stress patterns using StressMaker.

Pair of Units	T Diff (°C)	Workload characteristics of the stressmarks
L2 & I- Fetch	44.6	(1) Small data footprint and short local strides that result in high L1 D-cache hit rates and no stress on L2, and (2) 80% short-latency insns with high ILP and highly predictable branches – keeping fetch busy without pipeline stalls.
L2 & Register Remap	48.4	(1) 40% memory operations, large data footprint, and long local strides to miss in L1 and stress L2, and (2) 40% memory operations with very large dependency distances that put minimal stress on the register remap
L2 & Exec	44.4	(1) No memory operations, so no stress on L2, and (2) 40% short latency integer operations and 40% short latency floating-point operations that stress the execution unit.
Branch Predictor & L2	41.3	(1) 80% branches with transition rate equally distributed across all buckets (0-10%, ... , 90-100%) – a mix of difficult and easy to predict branches that stress the branch predictor, and (2) No memory operations, no stress on L2
Issue & LSQ	61.0	(1) 80% memory operations with small data footprint and short local strides stressing the load/store queue, and (2) limited activity in issue queue.

7. Related Work

Characterizing Power Consumption of CMOS circuits. A lot of work has been done in the VLSI community to develop techniques for estimating the power dissipation of a CMOS circuit. The primary approach in these techniques is to use statistical approaches and heuristics and to develop a test vector pattern that causes maximum switching activity in the circuit [8][16][22][24][27][28][33]. Although the objective of this paper is the same, there are two key differences compared to our work. First, our technique aims at developing an assembly test program (as opposed to a test vector) that can be used for maximum power estimation at the microarchitecture level. Second, developing stressmarks provides insights into the interaction between workload behavior and power/thermal stress, which is not possible with a bit vector.

Manually Developed Stressmarks. [11][12][13][34] refer to hand-crafted synthetic test cases developed in industry that have been used for estimating maximum power dissipation of a microprocessor. The Alpha Toast and Thumper hand-crafted stressmarks [11] stressed total power and dI/dt , respectively. In [23], stress benchmarks have been developed to generate

temperature differentials across microarchitecture units.

Tests for Performance & Functional Validation. Automatic test case synthesis for functional verification of microprocessors [3] has been proposed and there has been prior work on hand-crafting microbenchmarks for performance validation [4][9].

Statistical Simulation and Benchmark Synthesis. The primary objective of prior work in statistical simulation [10][25][26] and workload synthesis [1][17][20] is to reduce simulation time by cloning the performance of a program in a synthetic trace or benchmark, respectively. The key idea of these techniques is to capture the behavioral characteristics of a program execution in a statistical profile, and generate a synthetic trace or benchmark to reproduce the performance of the program. In contrast to this prior work, BenchMaker generates a synthetic benchmark from an abstract workload model consisting of a limited number of program characteristics. This enables exploring the workload space in search for stressmarks in the StressMaker framework.

8. Conclusions

Characterizing the maximum power dissipation and thermal characteristics of a microarchitecture is an important problem in industry. However, due to the complexity of modern microprocessors, and the need to construct synthetic test cases for various complex power and temperature phenomena, it is extremely tedious to manually develop and tune stressmarks for different stress criteria and microarchitectures.

In this paper, we developed BenchMaker, a framework for constructing parameterized synthetic benchmarks from an abstract workload model. One of the key results from this paper is that it is possible to fully characterize a workload from a limited number of microarchitecture-independent program characteristics, and still maintain good accuracy with respect to real workloads.

We subsequently leveraged BenchMaker by proposing a novel approach for automating the development process of stressmarks. StressMaker is a stressmark generation framework that uses BenchMaker and machine learning to automatically synthesize a stressmark from fundamental program characteristics by exploring the workload design space. We showed that StressMaker is very effective (1% deficiency) in constructing stress benchmarks for measuring max-power dissipation. And we provided case studies in which we constructed stressmarks for maximum average and single-cycle power

consumption, dI/dt stressmarks, temperature hotspot stressmarks and thermal stress patterns.

We believe StressMaker is a promising first step towards the automated generation of stressmarks. As part of our future work, we plan on evaluating StressMaker in an industrial environment and compare the stressmarks generated by StressMaker against manually developed stressmarks. Also, we will continue fine-tuning the abstract workload model in order to capture additional workload characteristics such as bit toggling in data values and instruction opcodes, as well as interactions between co-executing threads and programs in multi-threaded and multi-core processors.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable feedback. Ajay Joshi was supported by an IBM Fellowship. Lieven Eeckhout is supported by a Postdoctoral Fellowship with the Fund for Scientific Research in Flanders (Belgium). This work is also supported in part through the NSF award numbers 0429806 and 0702694, an IBM Faculty Partnership Award, the UGent-BOF project 01J14407, the FWO project G.0255.08, and HiPEAC.

References

- [1] R. Bell Jr. and L. John. *Improved Automatic Test Case Synthesis for Performance Model Validation*. In ICS, 2005.
- [2] Personal communication with Aparajita Bhattacharya (Senior Design Engineer) and David Williamson (Consulting Engineer), ARM Inc.
- [3] P. Bose. *Performance Test Case Generation for Microprocessor*. In the IEEE VLSI Test Symposium, 1998.
- [4] P. Bose and J. Abraham. *Performance and Functional Verification of Microprocessors*. In the IEEE VLSI Design Conference, 2000.
- [5] D. Brooks, V. Tiwari, and M. Martonosi. *Wattch: A Framework for Architecture-Level Power Analysis and Optimization*. In ISCA, 2000.
- [6] D. Brooks and M. Martonosi. *Dynamic Thermal Management for High-Performance Microprocessors*. In HPCA, 2001.
- [7] D. Burger and T. Austin. *The SimpleScalar ToolSet*, version 2.0. University of Wisconsin-Madison Tech Report #1342, 1997.
- [8] T. Chou and K. Roy. *Accurate Power Estimation of CMOS Sequential Circuits*. IEEE Transactions on VLSI Systems, 1996.
- [9] R. Desikan, D. Burger, and S. Keckler. *Measuring Experimental Error in Microprocessor Simulation*. In ISCA, 2001.
- [10] L. Eeckhout and K. De Bosschere. *Hybrid Analytical-Statistical Modeling for Efficiently Exploring Architecture and Workload Design Spaces*. In PACT, 2001.
- [11] Personal communication with Joel Emer, Intel, on the Alpha Toast (max power) and Thumper (dI/dt) stress tools.
- [12] W. Felter and T. Keller. *Power Measurement on the Apple Power Mac G5*. IBM Tech Report RC23276, 2004.
- [13] M. Gowan, L. Biro, D. Jackson, *Power Considerations in the Design of the Alpha 21264 Microprocessor*. In DAC, 1998.
- [14] S. H. Gunther, F. Binns, D. M. Carmean and J. C. Hall, *Managing the Impact of Increasing Microprocessor Power Consumption*. Intel Technology Journal, Q1 2001.
- [15] M. Haungs, P. Sallee and M. Farrens. *Branch Transition Rate: A New Metric for Improved Branch Classification Analysis*. In HPCA, 2000.
- [16] M. Hsiao, E. Rudnick, and J. Patel. *Peak Power Estimation of VLSI Circuits: New Peak Power Measures*. IEEE Transactions on VLSI Systems, 2000.
- [17] C. Hsieh and M. Pedram. *Microprocessor Power Estimation using Profile-Driven Program Synthesis*. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, 1998.
- [18] R. Joseph, D. Brooks, and M. Martonosi. *Control Techniques to Eliminate Voltage Emergencies in High Performance Processors*. In HPCA, 2003.
- [19] A. Joshi, *Constructing Adaptable and Scalable Synthetic Benchmarks for Microprocessor Performance Evaluation*, PhD thesis, The University of Texas at Austin, 2007.
- [20] A. Joshi, L. Eeckhout, R. H. Bell Jr., L. K. John. *Performance Cloning: A Technique for Disseminating Proprietary Applications as Benchmarks*. In IISWC, 2006.
- [21] D. Kanter. *EEMBC Energizes Benchmarks*. Microprocessor Report. July 2006.
- [22] C. Lim, W. Daasch, and G. Cai. *A Thermal-Aware Superscalar Microprocessor*. In ISQED, 2002.
- [23] K. Lee, K. Skadron, and W. Huang. *Analytical Model for Sensor Placement on Microprocessors*. In ICCD, 2005.
- [24] F. Najm, S. Goel, and I. Hajj. *Power Estimation in Sequential Circuits*. In DAC, 1995.
- [25] S. Nussbaum and J. E. Smith. *Modeling Superscalar Processors via Statistical Simulation*. In PACT, 2001.
- [26] M. Oskin, F. Chong, and M. Farrens. *HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Design*. In ISCA, 2000.
- [27] Q. Qui, Q. Wu, and M. Pedram. *Maximum Power Estimation Using the Limiting Distributions of Extreme Order Statistics*. In DAC, 1998.
- [28] S. Rajgopal. *Challenges in Low-Power Microprocessor Design*. In VLSI Design, 1996.
- [29] T. Sherwood, E. Perelman, G. Hamerley, and B. Calder. *Automatically Characterizing Large Scale Program Behavior*. In ASPLOS, 2002.
- [30] K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. *Temperature-Aware Microarchitecture*. In ISCA, 2003.
- [31] K. Skadron, M. Martonosi, D. August, M. Hill, D. Lilja, and V. Pai, *Challenges in Computer Architecture Evaluation*, IEEE Computer, 2003.
- [32] <http://www.spec.org/specpower/>
- [33] C. Tsui, J. Monteiro, M. Pedram, A. Despaigne, and B. Lin. *Power Estimation Methods for Sequential Logical Circuits*. IEEE Transactions on VLSI Systems, 1995.
- [34] R. Vishwanath, V. Wakharkar, A. Watwe, V. Lebonheur. *Thermal Performance Challenges from Silicon to Systems*. Intel Technology Journal, 2000.