Simulation of the Impact of NOOPS on CPU Temperature

by

Sameul Haque

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
December 12, 2015

Keywords: Simulation, Thermal Management, Power Management, Instruction Scheduling, Process Scheduling

Approved by

Sanjeev Baskiyar, Associate Professor of Computer Science and Software Engineering
Vishwani Agrawal, James J. Danaher Professor of Electrical and Computer Engineering
Xiao Qin, Professor of Computer Science and Software Engineering

Abstract

A C++ based simulation language is used to simulate instruction scheduling in which a set of processes is balanced by the ratio of memory operators. The simulation of the reduction of thermal profiles involves insertion of NOOPS in a simulated process buffer to store critical process paths using a novel path detection algorithm. The ratio of memory operators of a possible execution path associated with a series of simulated assembly instructions in a processor is simulated at run time using NOOPS to balance memory ratios over a simulated instruction sequence. A memory operator is any instruction which involves transfer of data between processor registers and main memory. The simulation uses a counter for calculating the number of memory operations in a process. Critical sections are stored in a buffer to improve efficiency and NOOPS are assigned to a process set in such a way that the processor never exceeds a predetermined ratio of memory operators over a simulated sequence for the purposes of power validation. The reduction of memory operators is formally verified in software using a high level language. It is verified that the ratio of memory operators is lower for a simulated process set and that NOOPS can be used to reduce CPU thermal profiles.

Acknowledgments

New graduate students are often under the impression that graduate coursework is similar to the work at the undergraduate level, that is, an MS Thesis is essentially extended class project. This is the primary error that I encountered when I started and I would like to thank my father Dr. Anwarul Haque, my thesis adviser Dr. Sanjeev Baskiyar and my professor Dr. Xiao Qin for putting me on the right track.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This research concerns high level simulations of energy aware process scheduling. A classical example where this would be useful is a client-server application on a mobile device which experiences dynamic memory load such as a web server or a database. The goal is to balance a process set by the rate of memory references such that processes with high memory reference rates do not interfere with other processes.

The goal is to do this at run time without any information provided about the processes provided by the operating system other than priority. In this model, processes on a device experiencing a high rate of memory references are slowed down. In this implementation, this is achieved using NOOPS. The safest characterization of the scope of this research would be simulating CPU-based load balancing of process memory references in real time. The reason we attempt to do this at the processor level as opposed to the operating system level is because operating systems provide no built in methods for determining rate of memory references of a process.

A high level description of the process scheduler is as follows. Processes are assigned time slices in a round robin scheduler, and whenever the ratio of memory references in a process buffer is exceeded, the processor schedules NOOP instructions until the rate of memory operators is low enough. The scope of this research is narrow. The scheduler is simulated using a high level language, but a full hardware implementation is not provided here. This research is oriented towards high level simulations of low level processes, that is, this work is intended to explore the feasibility of synthesizing low level hardware using C++.

The memory analyzer provides cycle accurate estimates of power usage, but the SystemC scheduler simulation is not intended to be used with an actual processor, but rather used to verify

that the process model can be used to simulate actual hardware though hardware design is not the goal.

The details of the simulation are as follows: The scheduling technique involves the detection of a critical section. A high level language is used to detect the critical section by detecting the outermost loop in a series of nested loops in a sequence of assembly instructions stored in the cache. The instructions stored in the buffer represent the current set of paths being executed by the processor. A reference to US Patent 7873820 is made for the architecture of a process buffer.

The next phase of this project involved verifying that the memory ratio is lower using this method and projecting thermal profiles for a 100 MHZ processor. Finally, NOOPS are actually inserted into CPU processes to determine temperature profiles. Simulation classes are used to generate the output of a set of scheduled processes. The outline of this is as follows: A FIFO scheduler is modified to execute processes in a circular fashion. During context switch, the critical path(s) associated with a process set is sent from the cache to the buffer if it is smaller than the size of the buffer. NOOPS are then scheduled to balance the ratio of memory references in the buffer. If there is no path smaller than the buffer associated with the process, the buffer may still be used for adding NOOPS.

This work involves converting a FIFO based model to a round-robin model and simulating algorithms and communication protocols between a process buffer and a scheduler. SystemC data structures are used to specify the process and NOOP instruction scheduling algorithms. This interface specifies the communication protocols for which processes are active and which processes are idle. The current model being used only two processes and assumes there is one core. Some constants are set such as the size of the buffer. The model works as follows: when a process is active it writes assembly instructions to the buffer stored in the scheduler. To balance the ratio of process memory references NOOPS are inserted into processes before they reach the buffer. In this model, it is assumed that each process has the same priority and that the time slice assigned by the scheduler is constant.

1.1 Literature Review

1.1.1 Experimental Validation of Instruction Power Models

The method Tiwari, Malik, Wolfe use to derive the base energy cost per instruction is discussed.[1] This method is as follows: A processor is typically divided into several pipeline stages. When every instruction in the pipeline is the same instruction, it is possible to use an ammeter reading to determine the base energy cost for that instruction. The central hypothesis presented here is that memory operations are the primary drivers of energy consumption and some experimental validation is done along this line. On the x86 architecture, the mov operation is indicated as a memory operation, but this is expanded in our definition to include any instruction that involves main memory to register transfer of data. Mov instructions are the instructions where this is formal definition of the instruction, but other instructions may implement this implicitly.

Large loops are created of the same instruction. When these loops are of sufficient size the contribution of the branch operand in the sequence will be negligible. In a pipelined CPU, several instructions are executed simultaneously. There is nevertheless a base cost per instruction which is derived from the energy cost of an instruction at each pipeline stage. This is derived by measuring the current during these loops and multiplying by the clock period, the voltage, and the number of cycles per instruction. It is not necessary to know how many cycles an instruction takes at each stage to derive the base cost per instruction but only the total number of cycles per instruction. The authors details the reason for this in their paper.

Using this methodology the authors derived a base energy cost per instruction for a 486DX2 processor. A main memory to register memory instruction is where this cost was high for this chipset, but it was not the highest. A main memory to register data transfer is implicitly defined in certain instructions and we may include these instructions in our definition of a memory operation.

In all cases, the variation in the base energy cost was found to be within five percent. Most variability is due to variation in operands associated with an instruction. This refers to instructions with the same type signature as well as same the lexical name. For instance, the energy of a

Table 1. Subset of the base cost table for the 486DX2 and the '934

	Intel 486DX2				Fujitsu SPARClite '934			
No.	**Instruction**	**Current** (mA)	**Cycles**	**Energy** $(10^{-8}J)$	**Instruction**	**Current** (mA)	**Cycles**	**Energy** $(10^{-8}J)$
1	`nop`	276	1	2.27	`nop`	198	1	3.26
2	`mov dx,[bx]`	428	1	3.53	`ld [%i0],%i0`	213	1	3.51
3	`mov dx,bx`	302	1	2.49	`or %g0,%i0,%l0`	198	1	3.26
4	`mov [bx],dx`	522	1	4.30	`st %i0,[%l0]`	346	2	11.4
5	`add dx,bx`	314	1	2.59	`add %i0,%o0,%l0`	199	1	3.28
6	`add dx,[bx]`	400	2	6.60	`mul %g0,%r29,%r27`	198	1	3.26
7	`jmp`	373	3	9.23	`srl %i0,1,%l0`	197	1	3.25

Figure 1.1: Instruction Energy Models

mov instruction with the same type signature may vary up to 3.5 percent depending on the specific operands used. The authors determine base costs associated with specific operand values by making measurements, but measurements of variability are not useful in program power estimation because operand values are determined at run-time.

In addition to the base costs associated with an operand values, there are other effects. For instance, when the same instruction is repeatedly executed, the base cost is measured differently than when we execute different instructions repeated because there is an overhead associated with changing circuit states. It is possible to compensate for this by including the overhead associated with change in circuit state in our calculation. When reading from the cache repeatedly, a greater variation in address values lead to 3 percent increase power usage. When writing data to the cache, variation in address values increased power usage by five percent. The overhead is small for the processor in question, but novel architectures may result in different values for circuit overhead.

Pipeline stalls can significantly affect the cycle per instruction count and reduce efficiency, but this effect is mediated by the use of NOOPS to prevent stalls. Reduction in pipeline stalls using NOOPS is a technique that is currently used in modern processors.

In addition to pipeline stalls, cache misses will result in cycle penalties. This penalty can be reduced by using a buffer to store the critical sections of process and keeping the instructions

associated with various function calls in the cache. A reduction in memory instructions will reduce cache miss rates through the reduction of memory operands. It is possible to simulate the reduction in cache misses through address space reductions in our work.

The authors verify that any instruction which uses memory operands will have high power cost whereas instructions which affect only register values will have a low power cost. It is possible to determine the power cost of a program by calculating the power of each instruction in the control path using the above techniques and including the values of the jump statements. It is assumed that there are no stalls or cache misses.

This technique is specifically to reduce stalls and cache misses but the author indicates these effects can be determined when doing program analysis for power estimation. The author includes a discussion on power consumption of the memory system. In addition to the cost on the processor of cache miss, the memory system incurs a cost of 1/2 to 1/4 of the cost of the miss on the processor depending on operand values.

The authors describe the energy usage of each component during the execution of a program specifically the CPU and the L1 cache. Most embedded processors are programmed directly in assembly and optimization is done by intuition. The human element of optimizing compilers can be removed via automatic optimization.

An optimizing compiler is the typical way we go about optimization of memory. Typically, optimizing compilers target high performance processors and not embedded systems. Manual optimization is typically done through the optimization of algorithms or optimization of data. Algorithmic optimization rewriting on the basis of intuition. Optimization of data is typically done through changing data formats to target an architecture.

Low level optimization is discussed and can be done in software or in hardware. A profiler is typically used for cycle accurate simulations. Procedures from the executable on a compiler are sent to a simulator. A software based simulator typically gives an overhead of 10 percent. Manual instruction level optimization typically results in a 30 percent decrease in energy in the sub-band synthesis algorithm. [2]

One technique mentioned in previous for reducing cache misses in previous literature is reordering instructions such that the cache miss rate will be lower without any reduction in the ratios of power intensive instructions. The algorithm finds a minimal cost instruction sequence on the basis of cache miss rates using a directed acyclic graph. This is a compiler level optimization and can be implemented without any hardware modifications. The instruction graph used is similar to the instruction graph we use to detect the critical sections of a process, but it is implemented by the compiler whereas our technique is implemented using circuit logic by the analysis of branch operands. Compiler based approaches to instruction reordering are effective in reducing cache misses, but existing programs would need to be recompiled to use such techniques though instruction reordering does not seem to be as beneficial as memory operator reduction. [3]

The instruction power model is verified through simulation. The physical model of the processor is primarily based on previous work, but nevertheless an attempt is made to verify it. Previous work indicates that instruction count based approaches are within five percent of actual energy consumption.

More than just the cycle per instruction count to determine the energy usage for a particular instruction. These energy models are based on physical models of interconnects and pins. After reading of previous literature, it was determined that the highest power usage per instruction is typically associated with an L2 cache miss because the time and energy penalties are accrued from both the L2 cache and main memory accesses. This is primarily the reason that reductions in the number of memory operands will reduce energy consumption because cache hit rates will be lower. This physical model is the theoretical basis for our work on energy optimization via the reduction of memory operators because reductions in the rate of memory operators would necessarily result in the reduction of memory operands over any time interval. The physical model simply verifies that these operands are intensive thermally. Some experimental verification is done that these instructions are thermally intensive.

The specific scheduling optimizations mentioned may not take into account cache hit rates. That is, a scheduled processes sequence could potentially have a higher energy expenditure if the

cache hit rates are lower. This effect would be mediated by a buffer and it would also be mediated specific regions of the cache were allocated to each process. This is area where it is possible to investigate further. Even if cache hit rates are not considered, instruction count based approaches still appear to be within five percent of energy consumption even if the model is not valid in special cases.

An analysis is included of what pattern of memory accesses are most likely to increase processor temperatures. Operators which involve reads or writes to memory have higher current values and often have higher cycle counts which multiplies the energy cost of the instruction. Caches misses, misaligned accesses, and stalls can all increase the cycle counts of a memory access.

The energy of the memory system also contributes to the cost of the program. Consequently a program with a high number of memory accesses will be the most intensive in terms of program cost. Tiwari Malik and Wolfe indicate that compilers can be used for this purpose as well as elements inside of the processor through better use of registers. Compilers which make optimal use of registers show 30-40 percent reduction in energy. Though more than cycle count of a program is necessary to determine the energy cost of a program, optimization can be done through cycle counts which are usually greater for operations with memory operands.

Tiwari, Malik and Wolfe indicate circuit state changes may add overhead to a program that might contribute to program energy cost. [1] Therefore, instructions which repeatedly change circuit states may incur a penalty. From this experimental work, the most energy intensive code appears to be memory reads or writes over a very large address space.

1.1.2 Source Code Profiling

The next review topic concerns source code profiling. [2] For portable appliances, there is typically a commodity components with a microprocessor based architecture. For these components, FPGA hardware may be used for debugging.

There are some prototype tools used to estimate energy consumption of processor cores, caches and main memory in a SOC design. The final system energy is obtained by summing

the energy consumed by the execution of each instruction. Estimation of energy costs was within five percent of the measured range.

Code optimization is the process of translating high level specifications in to machine code suitable for optimization. An optimizing compiler targets high performance dedicated processors typically not embedded processors. The author indicates that the reduction of memory operands can be used optimize energy profiles. This is the theoretical basis for our research.

Since support for optimizing compilers is limited, the author focuses on manual rewriting of code. Specifically the profiler focuses on rewriting and optimizing code. This allows designers to focus on an abstract view of the problem, find good solutions, then move down in abstraction.

The top layer mostly focuses on algorithms. The original specification is profiled to determine where the most time and power are spent and the author determines alternative algorithms.

At a lower level of abstraction, the focus is on changing the representation of manipulated algorithms. The main objective is to match characteristics of target architectures with processed data. Signal processing algorithms are often hard to specify.

The total amount of energy to be consumed was plotted on cycle by cycle plots. The compiler was mostly rewritten manually. The improvement in terms of energy consumption of a manual optimization resulted in a 20 percentage reduction in the net amount of power used for the program.

Furthermore, instruction level optimization resulted in a 30 percent decrease in energy use. As it concerns extending the life of device during operation these findings are significant.

The hardware level design tools used can often be used to evaluate quite many revisions of source code. The CPU can be changed to ARM based processor. Burst SDRAM will also increase power usage. These simulations were run with modifications made to the compiler during the course of development. The speedup registered by the optimized compiler was not necessarily representative of real world situations.

Simunic, Beneni, and Micheli propose several techniques for reducing energy consumption such as reducing sequential loads with a single load expression. [2] They also propose storing finding common sub-expresssions and storing them in register values. Both these methods are compiler

level optimizations and can be implemented in software and constitute methods for reducing the number memory references in a program.

1.1.3 A Review of Process Management

Consider existing systems for balancing multiple applications. A multilevel feedback queue is used with higher priorities assigned to short processes and I/O bound processes. There are only a subset of processes which can be balanced using the techniques mentioned here. Specifically, I/O processes, extremely short processes or system level processes are excluded. Typically, video or audio processes are assigned higher priorities so as not interfere with their performance. The process management technique will be used to balance user level processes that have no priority or the same priority. This will be done extracting information from an extended loop buffer regarding cyclical paths and the percentage of memory operations occurring in the loop over the duration of the loop.

When it is determined that a path will be cyclical, it is possible to disable portions of the cache.[5] This is not simulated, but the path detection and NOOP insertions are is simulated. It is possible to use a buffer to insert NOOPS in between memory references to reduce thermal profiles whether or not a path is cyclical. It is possible calculate the number of memory operations occurring for a duration of the path using a counter. This process model is simulated in Java and then translated into a C++ based simulation language.

The basis for the hardware simulation is a instruction queue. A CPU loop buffer pre-fetches an instruction sequence and if the target of a conditional branch is contained within the process buffer then the instruction is fetched from the buffer. Loop buffers are currently used in ARM Cortex chips which are used in many advanced electronic devices. This loop buffer is to determine the ratios of memory operations occurring in an execution path. It is possible to disable the instruction cache when a path is cyclical and insert NOOPS for validation purposes regardless of whether a path is cyclical.

Future research could involve the use of better simulation software to get more accurate energy profiles. For internal use, a model has been developed where assembly instruction sequences are simulated in processors, but the energy profile of each instruction was not determined for a given chipset.

Chapter 2

Problem Statement

The problem described in this research concerns real time scheduling as it applies to thermal aware computing. It is necessary to establish hard constraints on energy and thermal profiles as this is the a critical factor in determining whether a processor will be commercially viable. Typically, the methods for determining when a processor will use energy involves measuring total operations per second or measuring run time temperature. These methods usually after processors have already exceeded some tolerance. This work involves trying to determine the behavior of the processor at runtime using instruction ratios.

A good understanding of process structure is developed and how it might be used to determine the rate memory operators of execution paths in real time. The primary area of research analyzed was branch and jump statements as these are typically what are used in execution paths. It was determined how to detect memory references rates of an execution paths in code given a sequence of assembly instructions.

In an assembly path, conditional branches or jump statements execute with respect to a test condition. Given an assembly path, one can determine the number of memory operations in the path and its total number of instructions. A more complex case developed involved nested paths and included function calls.

System C is a set of C++ classes that allows us to use C++ as a hardware description language. There is some syntactical overhead compared to Verilog and VHDL. System C programs are larger. However, this greatly outweighed by the relative ease with which existing software algorithms can be implemented in hardware. It can also be used to simulate concurrent processes.

When tasks are scheduled to minimize power consumption or thermal radiation, it is typically done by assessing the profiles of processes experimentally. This work is concerned with determining the ratio of memory operations occurring in hardware and assigning NOOPS at runtime in hardware. The critical insight is to assign priorities according to thermal profiles determined from run time instuction level analysis.

In real world performance scenarios, memory related performance issues are the primary drivers of power and thermal radiation due to L2 cache misses. The goal is to reduce power consumption by ensuring that the system always stays within the thermal profile assigned to it. The proposal is to create a novel thermal scheduler such that one can perform run time analysis on the power profiles of running processes by determining the structure of execution paths. Then, it is possible to simulate the assignment of NOOPS to running processes such that a thermal intensive process will run slower.

The applications of this research are in the area of mobile computing and in the data centers that utilize low power hardware such as ARM chips. The benefits are also in increasing the cost competiveness of data centers provided that the algorithms are implemented by the Operating System. There is an across the board reduction in power consumption due to a reduction in cooling costs.

There are also increases in system lifespan. The initial cost for purchasing datacenter hardware is surpassed by the energy requirements needed to efficiently run the datacenter within 2 to 4 years depending on the datacenter size. The hardware needs to maintain a certain temperature to insure integrity, thus cooling systems are used which also increases the energy cost. The algorithms we discuss here primarily have benefits in the area of mobile computing.

This project focuses on energy management using a combination of software and hardware based techniques, namely load balancing algorithms for memory operations which are the primary drivers of energy consumption according to previous work. [4] Memory operations require far more energy and produce more heat than arithmetic instructions. This becomes more apparent if a cache miss occurs and data must be retrieved from main or virtual memory. By keeping the ratio of

memory operations below a predetermined value, it is possible to use less energy and dissipate heat more efficiently from those components. Thus, energy usage is reduced, less heat is generated, and the environmental impact of datacenters is reduced.

Priorities are assigned in hardware such that a predefined ratio of memory operations is never violated. First, the maximum ratio of memory operations that may occur on a particular hardware is determined. A counter is used to determine the memory ratio of each process executed in a round robin scheduler.

The processor is first analyzed in software and then then modeled into a hardware design. The solution is evaluated using assembly level analysis tools. Assembly sequences of compiled programs are analyzed using a freely available tool known as IDA-pro and and an internally developed process analyzer we discuss here.

This technique differs from standard memory management techniques because one is not concerned with the amount of memory allocated but the ratio of memory operations that are occurring in the processor over a specified instruction sequence. If one can assign processes such that a constant number of memory related instructions always occurs, then one can be fairly sure that our system stays within its thermal profile except for certain highly specialized applications that perform a large number of floating point operations on a small set of memory addresses. This condition is rare.

The solution is verified by showing that for usage profiles associated with a typical program that the ratio for the maximum number of memory instructions over a specified is not violated. This is an area of further research. Typically, scheduling and the assignment of priorities is something that occurs at software layer therefore one would have to include some additional hardware that transmit memory ratio information to the control registers on the processor. For the purposes of this project we are using the ARM documentation. The software model will specifically refer to the ARM priority structure.

This work is mostly concerned with client side memory balancing not server side memory balancing. However, algorithms can be applied to ARM based servers. This research does not

necessarily involve any specific mobile platform. Nevertheless, when one models the memory ratio of our hardware device we will specifically refer to the x86 architecture. The ARM architecture is used on units which run the iOS operating system, but there are also inexpensive Linux PCs which also use ARM such as the recently released Raspberry Pi. This research has specific applications in the balancing of memory operations when multiple applications running in these types of devices.

Operating systems typically ship with advanced scheduling algorithms and memory management. This research attempts build on these technologies. Specifically, the goal is to balance memory ratios on operating systems which run multiple applications such that thermal constraints are met. This is done by ensuring that a constant ratio of memory operations is not violated for a specified time interval.

Chapter 3

Simulation

3.1 Software Models Of Hardware

The algorithms are implemented in software using a standard C++ compiler. Microsoft Visual Studio and Eclipse are both used. The algorithm to detect a path is coded in Java. The scheduler simulation is coded in using an extension of C++. All sources of algorithms we utilized for simulations are noted. The majority of the code base is completely original. The complete hardware implementation is not documented here.

The language used to design the hardware is a derivative of the C++ language, System C. It was chosen specifically because of how easy it is to take existing software based algorithms and implement them in hardware. The specific role here is to determine in real time when the processor will use a large number of memory operations. This is not done using statistical methodologies as is done with typical branch prediction algorithms but by anticipating the memory operation rates of execution paths, and then inserting NOOPS into processes based on these structures using a buffer.

.

The instruction sequences that were analyzed were not randomly generated with the process analyzer were not randomly generated. To get accurate data, what was instead used were assembly sequences of compiled programs. This was done in the case of an x86 program. To do this for ARM, it would be necessary to use the ARM development kit in combination with source code analysis tools, or it would be necessary to develop our own source code analysis tools. The process scheduler can in fact be used with processes that have different priorities. Only a cursory look is taken at such a design, and instead processes are emphasized that have the same priority or no priority.

The unmodified C++ based processor simulation software has the following output for a generic RISC processor:

```
IFU : mem=0xf550000
IFU : pc= 19 at CSIM 173 ns

-
ID: R5=R5(=5)
: at CSIM 175 ns
-

-
ALU : op= 3 A= 5 B= 0
ALU : R= 5-> R5 at CSIM
177 ns

-
-
ID: R5=05(5) fr ALU at CSIM 178 ns
-

IFU : mem=00
IFU : pc= 1a at CSIM 180 ns

-

*******************************
ID: REGISTERS DUMP at CSIM 182 ns
*******************************
REG :================================================
R 0(00000000) R 1(00000001) R 2(fff000e2) R 3(ffffffff)
R 4(00000004) R 5(00000005) R 6(0000000a) R 7(fcf0fdef)
R 8(00000008) R 9(00000009) R10(00000010) R11(0000ff31)
R12(0000ff12) R13(00000013) R14(00000014) R15(00000015)
R16(00000016) R17(00fe0117) R18(00fe0118) R19(00fe0119)
R20(00fe0220) R21(00fe0321) R22(00fe0322) R23(00ff0423)
R24(00ff0524) R25(00ff0625) R26(00ff0726) R27(00ff0727)
R28(00f70728) R29(00000029) R30(00000030) R31(00000031)
```

Figure 3.1: Unmodified Processor

When dealing with a hardware logic structure as opposed to a software logic structure, often additional constraints exist regarding the logic functions that may be used. For instance, if it is needed to calculate the mean of a set of values in real time, it is simple to do using hardware logic, but calculating the mode is generally much more difficult. The goal is to assign NOOPS in a scheduler to user level processes of the same priority or no priority in real time by determining their memory ratios. On the software side, it is possible to determine memory ratios by assuming that it is a constant fraction of the CPU utilization of a particular process. Once it is calculated that a cyclical path will be small enough for the instruction buffer, the instruction cache may be disabled data is transferred to the buffer. [5]

The next goal was to develop software models that will accurately simulate the process models that have been developed. The model has been developed well enough to detect nested structures, context switches have been modeled, and constants have been associated with the buffer sizes in software. These were immediate priorities. Real assembly sequences associated with actual processes as opposed to simulated processes were used in our path analyzer. A hardware simulation was using a C++ based hardware description language. The language we used to specify hardware architecture is a set of C++ classes known as SystemC. Future research may involve determining power usage associated with the logic structures using better simulation software. The final task was to verify conditions for which our process model maintains thermal and energy constraints and the conditions for constraints are violated.

3.2 Assembly Data From Apache.exe

Initially, this work was mostly concerned with learning how to use the IDA disassembler and transferring executable data from an executable file to the simulator. The raw data from an Apache WebServer process is provided here. This is the data we read into our analyzer.

Raw assembly from a sample path in apache.exe reads as follows:

A sequence of x86 instructions is read into the Java based analyzer. The file that was read was the Apache WebServer. This is an example of a background process with dynamic memory

```
int nestedloops = 0;
int loops = 0;
int zold = 0;
for (i = 0; instructions[i] != null; i++) {
    if (instructions[i]=jump,branch) {
      for (int z = 0; z <= i; z++)  {
    if (target[i]==address[z])) {
loops++;
if (z < zold) {
    nestedloops++;
Print ("Nest: Outer:" + address[z] + " Inner: " +   address[zold]);
                              }
     zold = z;
     Print("Loop Start: " + address[z]);
     break;
 }
      }
}
}
```

Figure 3.2: Pseudocode For Path Detection Algorithm

```
push ebp
mov ebp, esp
sub esp, 30h
push ebx
push esi
lea eax, [ebp+argv]
push edi
lea ecx, [ebp+argc]
xor esi, esi
push eax
push ecx
mov [ebp+var_18], esi
mov [ebp+var_24], 404170h ;
mov [ebp+var_20], 404168h ;
```

Figure 3.3: Beginning of Apache.exe Raw Instruction Path

rates where the memory balancer would be applicable. The next steps is to convert the string base instructions into data structures and determine a base memory rate based on all instructions in the sequence.

The assembly sequences from files and converted them into data structures. The analytical software used to read apache.exe actually displays cycles and code trees in a comprehensive way. These cycles are reconstructed in a software analyzer insofar as they allow us to determine the memory rate of an execution path in hardware. The first step is to reconstruct code cycles determined from the assembly analysis tools.

After using the disassembler and tracing the execution path of the processor, it was possible to determine precisely the sequence of instructions going to the processor for a sample execution path. The goal of the analyzer is not to determine the memory rate from the instructions of a compiled program but to determine the memory ratios from instruction sequence inside the processor. Though this work was successful in reading an assembly sequence from compiled programs, in order for the program to determine the execution paths associated with path cycles, these paths must be manually reconstructed inside the processor. The next step is to calculate the memory ratios for a sample path using our analyzer. Function calls and methods to handle differing execution paths are discussed

Below is a graph of instruction paths inside the processor. When the processor executes a cyclical path, a path similar to the one below is stored in a buffer: [5]

Once the assembly sequence was reconstructed, the sequence was read into the analyzer to determine memory reference rates.

When actual assembly sequences were analyzed, it was determined that the model used to describe assembly sequences was largely accurate. However, some modifications were made to our predictive model to handle real world data. An approximate value was determined for the memory usage associated with functions being called. However, this was done manually. In order to determine exact memory ratios, code cycles were reconstructed with function calls included as was done for cycles without function calls included. This was a manual process.

```
Reconstructed Assembly Sequence:

loc_401116:
movsx eax, byte ptr [ebp+var_1]
add eax, 0FFFFFFC1h
cmp eax, 37h
ja loc_4013FD

xor edx, edx
mov dl, ds:byte_401818[eax]
jmp ds:off_4017D4[edx*4]

loc_40115A:
mov ecx, ds:ap_server_pre_read_config
mov edx, [ecx]
push edx
call ds:_apr_array_push@4 ; apr_array_push(x)
mov ecx, [ebp+var_14]
mov esi, eax
mov eax, [ebp+var_8]
push eax
push ecx
call ds:_apr_pstrdup@8 ; apr_pstrdup(x,x)
mov [esi], eax
jmp loc_4013FD

loc_4013FD:
mov ecx, [ebp+var_1C]
lea edx, [ebp+var_8]
lea eax, [ebp+var_1]
push edx
push eax
push offset
push ecx
call ds:_apr_getopt@16 ; apr_getopt(x,x,x,x)
test eax, eax
jz loc_401116
```

Figure 3.4: Apache.exe Simulation of Hardware Path Reconstruction

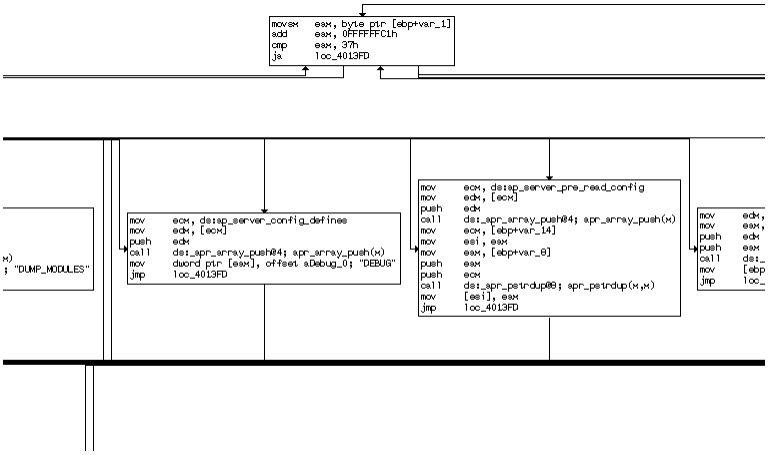

Figure 3.5: Apache.exe Function Graph

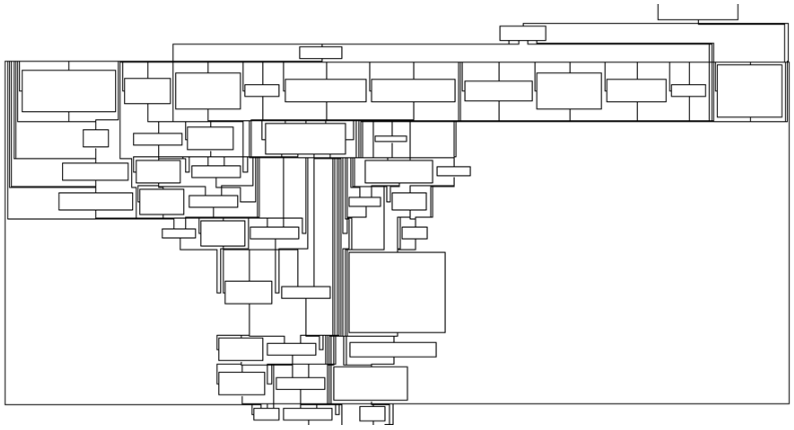

Figure 3.6: Expanded Apache.exe Function Cycle

Keep in mind that process buffer sizes were selected in the analyzer such that it was possible to determine the memory ratios of 10e9 instructions in real time. In order to do this, instructions are stored into the buffer until there is a sufficiently small path. One is able to determine memory rates in real time as follows: when path(s) are cyclical and small enough and to store in the buffer, one may assign NOOPS to a set of processes to the balance memory ratios.

3.3 Simulation Output

When real assembly sequences were analyzed, it was determined that the model used to describe assembly sequences was largely accurate. However, some modifications were made to the predictive model to handle real world data. It was possible to get an approximate value for the memory usage associated with functions being called. However, this was done manually. In order to determine the exact memory ratios, code cycles were reconstructed with function calls included as was done for cycles without function calls included. This was a manual process.

Keep in mind that a buffer size is selected in the analyzer such that it is possible to determine the memory ratios of 10e9 instructions in real time. In order to do this, instructions are read into the buffer until a sufficiently small path was reached. It is possible to assign memory rates in real time as follows: when path(s) are small enough and cyclical, NOOPS may be assigned to the process to the processes inside a buffer balance memory ratios.

Below is a manually reconstructed assembly path for apache.exe on x86 with function calls included:

```
loc_401116:
movsx eax, byte ptr [ebp+var_1] add eax, 0FFFFFFC1
cmp eax, 37h
ja loc_4013FD
xor edx, edx
mov dl, ds:byte_401818[eax]
```

```
jmp ds:off_4017D4[edx*4]
...
...
...


loc_6EED1B1F: mov edx, [ebx+8]
mov ecx, [ebx+10h] inc edx
mov eax, edx mov [ebx+8], edx dec eax
imul eax, [ebx+4] add eax, ecx
pop ebx pop ebp retn 4
_apr_array_push@4 endp
mov ecx, [ebp+var_14] mov esi, eax
mov eax, [ebp+var_8] push eax

push ecx
call ds:_apr_pstrdup@8 ; apr_pstrdup(x,x)
 public _apr_pstrdup@8 _apr_pstrdup@8 proc near
push ebp mov ebp, esp push esi
mov esi, [ebp+arg_4] xor eax, eax

test esi, esi
jz short loc_6EED0718

-------

loc_6EEC90EF:
```

```
cmp dword ptr [esi+10h], 2Dh
jz loc_6EEC8FF3


loc_6EEC8FF3:
mov ecx, [ebp+arg_8] mov al, [esi+10h] pop edi
pop esi
mov [ecx], al mov eax, 1117Eh pop ebp
retn 10h
test eax, eax
jz loc_401116
```

The next step was reading in the assembly sequences into the memory analyzer from a reconstructed path for simulation of hardware based determinations of memory rates. For practical implementation, the detection of paths were restricted to memory locations inside the currently executing process.

3.4 Simulation of Hardware Based Path Detection in Java

3.5 Temperature Projections

The next step involved formal verification of NOOP insertions for the purpose of power simulation. Verification can be a very time consuming process and this work was largely in the initial phases. A FIFO scheduler was modified to work instead as a round robin scheduler, inserting NOOPS into processes on the basis of the process model from the Java based algorithm.

The design specification specified was that the memory ratio would be lower using this technique. This was verified. In our simulation, the memory ratio of an unscheduled sequence was 0.289 whereas it was 0.189 for the scheduled sequence for a two process set with a maximal memory ratio of 0.2.

```
97 memory
257 instructions

movsx eax, byte ptr [ebp+var_1]
add eax, 0FFFFFFC1h
cmp eax, 37h
ja loc_4013FD

xor edx, edx
mov dl, ds:byte_401818[eax]
jmp ds:off_4017D4[edx*4]

add     eax, eax
mov     [ebp+arg_0], eax
mov ecx, ds:ap_server_pre_read_config
mov edx, [ecx]
push edx

call ds:_apr_array_push@4 ; apr_array_push(x)
call    _apr_palloc@8   ; apr_palloc(x,x)
retn    8
retn    4

...
...
...

jz loc_401116
Loop Start: loc_6EED1ABE
Loop Start: loc_6EEC77B8
Loop Start: loc_6EEC77DA
Loop Start: loc_6EEC77FE
Loop Start: loc_6EEC79B8
Loop Start: loc_6EEC79C6
Nest: Outer:loc_401116 Inner: loc_6EEC79C6
Loop Start: loc_401116
Total Loops: 7
Nested Loops: 1
```

Figure 3.7: Output of Nested Path Detection Algorithm

What was not verified was that the actual energy profile of the processor would be lower, only that the memory ratio would be lower.

The last phase of this project involves verification of power constraints for several path sets. A RISC processor design implemented using SystemC was simulated, but there were difficulties in determining the energy profiles. It is feasible to validate the processor model and some exploratory work was done along this line.

This typically involves simulating the NOOP insertions using real assembly sequences, but it is not required. For these purposes, real assembly sequences were not used for the verification of NOOP insertions. There has already been validation work done for the detection of critical sections of code by using a real sequence on the path detection algorithm though we did not do any verification work on the implementation this section as this is beyond our current scope.

The verification work involved converting the FIFO based model to a round robin model and specifying methods for communication between the processor and a process buffer. Some constants have been set such as the maximum number of processes to be 150. The model works as follows when a processes is active it writes assembly instructions to the buffer and from these instructions we determine the ratio of memory references as a proportion of the total number of memory instructions which are allowed over the interval.

Without NOOP insertions, the memory ratio is 0.28 which is significantly higher than the 0.189 ratio for a scheduled instruction sequence with a maximal memory ratio of 0.2. Given that memory related instructions are the primary drivers of energy consumption, our hardware algorithm will automatically optimize a process set to keep the memory ratio below the specified value. Previous work indicates that the reduction of memory operands can be used to optimize power consumption. [1] Our work falls into line of automatic optimization on this basis. NOOP Ratios are determined from a C++ based simulation. From the C++ based simulation, projections are made of the number of baseline instructions, number of memory instructions, and NOOP instructions in a critical path by multiplying the ratios from the simulation by the size of the buffer.

Maximal Ratio of Memory References	Ratio of Memory References from Simulation	Total Instructions	Number of Memory Instructions in Scheduler	Number of NOOPS	Length Penalty Percent
Unscheduled	0.242	141	34	0	0
0.15	0.150	206	31	65	146
0.20	0.178	202	36	42	126
0.25	0.187	171	32	25	117
0.30	0.207	187	32	25	109

Table 3.1: Projected Instruction Ratios For Varying in Maximal Memory Instruction Count On A Simulated Process Scheduler - Path Set 1

From this power for the path is calculated, adding up the power cost of each instruction from a formula derived from the instruction energy values mentioned by Tiwari:

$$PathPower = (3.69 * MemoryIns + 2.5 * BaselineIns + 2.26 * NOOPIns) * 10e - 8J \quad (3.1)$$

The energy for a 486DX2 processor at 100MHZ is then projected from the formula:

$$Energy = PathPower/(NumberOfInstructions/ClockRate) \quad (3.2)$$

A temperature projection is made for a newer processor model by assuming that the reduction in wattage will be in the same range in newer processor models. [6]:

$$Temperature = 0.29 * Energy + 45.1 \quad (3.3)$$

The work for process power validation was mostly exploratory. The goal is to determine the power dissipation in watts of a series of scheduled instructions. A successful effort was made in acquiring the cycle count for a set of x86 instructions, but could not determine the power dissipation from this cycle count. To determine the power dissipation of a scheduled instruction set, one needs to determine more than just the cycle counts of an instruction set, one needs to determine the

Maximal Ratio of Memory References	Ratio of Memory References from Simulation	Total Instructions	Number of Memory Instructions in Scheduler	Number of NOOPS	Length Penalty Percent
Unscheduled	0.42	128	54	0	0
0.15	0.15	317	48	197	264
0.20	0.20	239	48	119	199
0.25	0.25	196	49	74	160
0.30	0.3	160	48	40	133

Table 3.2: Projected Instruction Ratios For Varying in Maximal Memory Instruction Count On A Simulated Process Scheduler - Path Set 2

Maximal Ratio of Memory References	Ratio of Memory References from Simulation	Total Instructions	Number Instructions in Scheduler	Number of NOOPS	Length Penalty Percent
Unscheduled	0.261	141	34	0	0
0.15	0.151	172	26	66	162
0.20	0.175	171	30	53	144
0.25	0.207	140	29	25	121
0.30	0.235	136	32	12	109

Table 3.3: Projected Instruction Ratios For Varying in Maximal Memory Instruction Count On A Simulated Process Scheduler - Path Set 3

NOOP Ratio	Baseline Ins.	Memory Ins.	NOOP Ins.	Cost 10-8J	Energy (W)
0	1344	726	0	6038.9	29.1
0.31	1620	450	930	7747.5	25.8
0.207	1845	534	621	7986	26.6
0.146	2001	561	438	8062	26.9

Table 3.4: Projected Power Data For Buffer at Varying NOOP Ratios - Path Set 1

NOOP Ratio	Baseline Ins.	Memory Ins.	NOOP Ins.	Cost 10-8J	Energy (W)
0	658	476	0	3406	29.9
0.62	685	450	1864	7588	25.3
0.49	906	600	1493	7855	26.2
0.377	1117	1132	750	8120	27.0
0.25	1350	750	900	8391	28.0

Table 3.5: Projected Power Data For Buffer at Varying NOOP Ratios - Path Set 2

power dissipation for the instruction on the basis of processor models mentioned in previous work. [1] This is an extensive area of research. In previous work, the model was developed using an ammeter. It was unclear whether power dissipation from the scheduled instruction sequence can be determined from this software library without extensive additional work derived from the original paper.

The outline of this is as follows: from a processor model, the average power dissipation of each instruction is determined. Then, the total power dissipation is determined by summing the power dissipation of each instruction. The scheduled NOOP ratios and associated power and temperature values are as follows:

NOOP Ratio	Baseline Ins.	Memory Ins.	NOOP Ins.	Cost 10-8J	Energy (W)
0	1366	482	0	5196	28.1
0.388	1395	453	1151	7762	25.8
0.309	1545	525	929	7901	26.3
0.178	1843	621	535	8110	27.0
0.085	2030	705	264	8275	27.5

Table 3.6: Projected Power Data For Buffer at Varying NOOP Ratios - Path Set 3

NOOP Ratio	Projected Energy at 100MHZ (Watts)	Projected Temperature at 100MHZ (C)
0	29.1	53.56
0.31	26.0	52.65
0.207	26.6	52.82
0.146	26.9	52.89

Table 3.7: Projected Power Temperature and Temperature Data for Varying NOOP Ratios

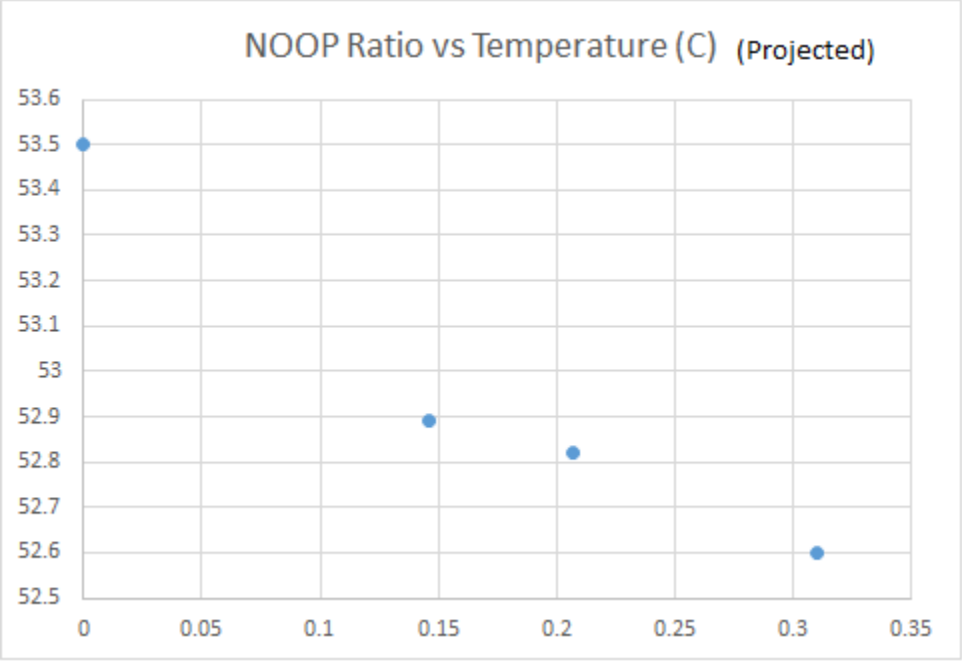

Figure 3.8: Projected Temperature Reduction: NOOP Ratio vs Temperature - Temperature = 0.29*Energy(Watts) + 45.1

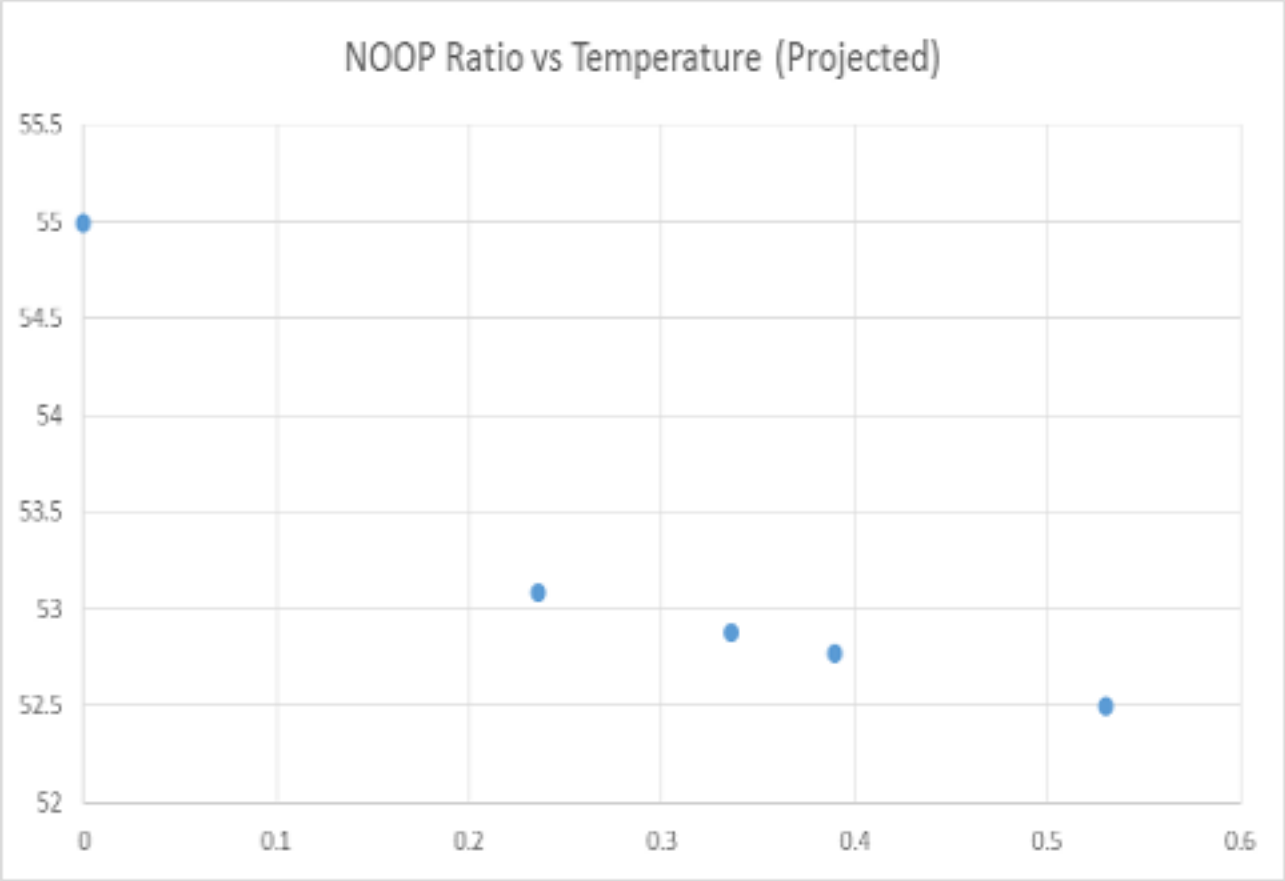

Figure 3.9: Projected Temperature Reduction: NOOP Ratio vs Temperature - Temperature = 0.29*Energy(Watts) + 45.1

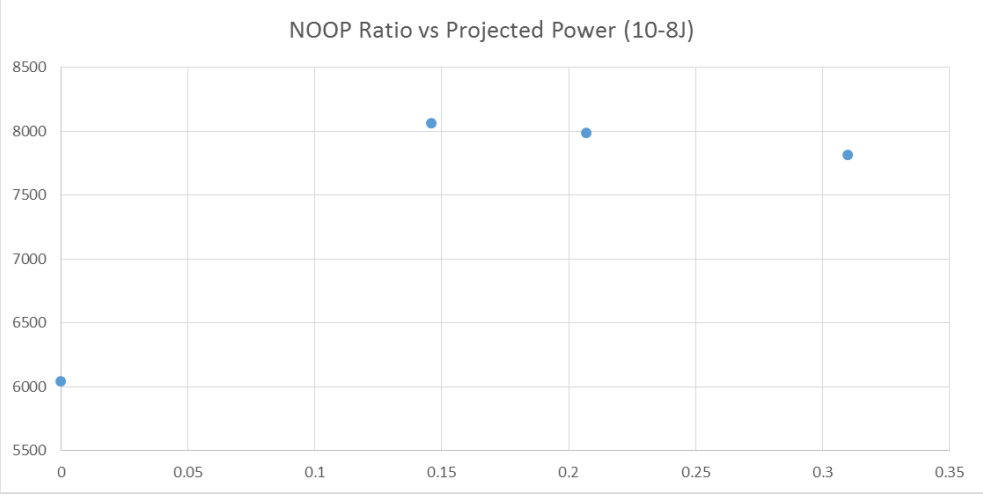

Figure 3.10: Projected Power Difference of Buffer Execution: NOOP Ratio vs Power

3.6 Temperature Reduction Results From NOOP Insertions

Fig 3.8 and 3.9 provide projected temperature reductions from two different simulations. 3.10 provides projected power data. The projected reductions are compared to reductions from three different simulations. The simulation data below provides time vs average core temperature graphs and peak and average core temperature bar charts for the Lattice Boltzman Method and Shor's Factorization Algorithm. Simulation data from the sjeng chess simulation program and the srand simulation is also provided in low NOOP ranges. The NOOP ratio was not measured to be high for the sjeng simulation. Peak and Average temperature data tables are provided for Lattice Boltzman Method, Shor's Factorization Algorithm and sjeng chess simulation. The peak temperature in each of the following simulations corresponds closely to the projected temperature values as this is close to the limiting temperature. For the sjeng chess simulation program, comparisons are provided for peak temperature data in a Windows Desktop environment. It is established using the central limit theorem used in statistics that the average processor temperature over a simulation is a normally distributed measurement.

3.7 Lattice Boltzman Method

The Lattice Boltzman Method:

The Lattice Boltzman Method is used to simulate incompressible fluid flows by solving the Boltzman equation which is a linear partial differential equation used in statistical mechanics to describe many particle systems.

df/dt=(df/dt)force + (df/dt)coll + (df/dt)diff

NOOPS are into a portion of the code used to simulate streaming collisions in this particle system. Figure 3.11 and 3.12 indicates isolated core temperature reductions for the Lattice Boltzman Method on Laptop AC and Laptop Battery.

Fig 3.13 - Fig 3.14 indicate the differences in the average temperature for all cores for varying number of NOOPS for the Lattice Boltzman Method on Laptop AC and on the Linux Desktop. Fig

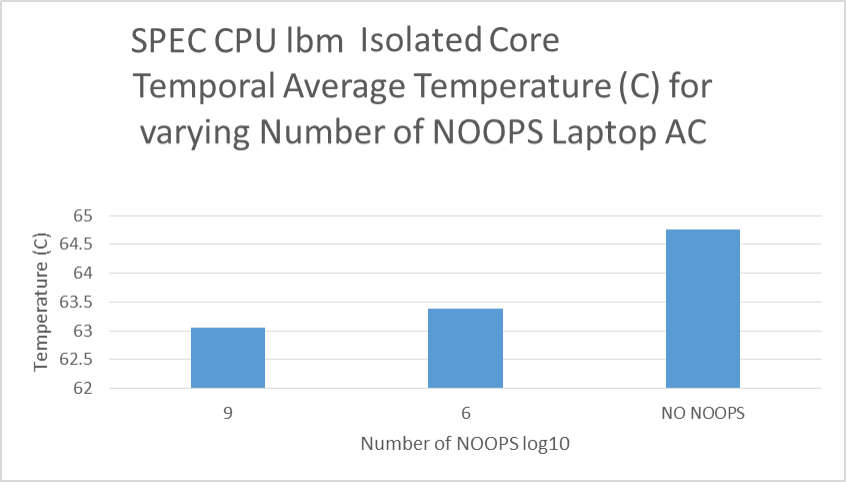

Figure 3.11: SPEC CPU lbm Measured Temporal Average Temperature for Isolated Core (C) for varying Number of NOOPS Laptop AC - Linux

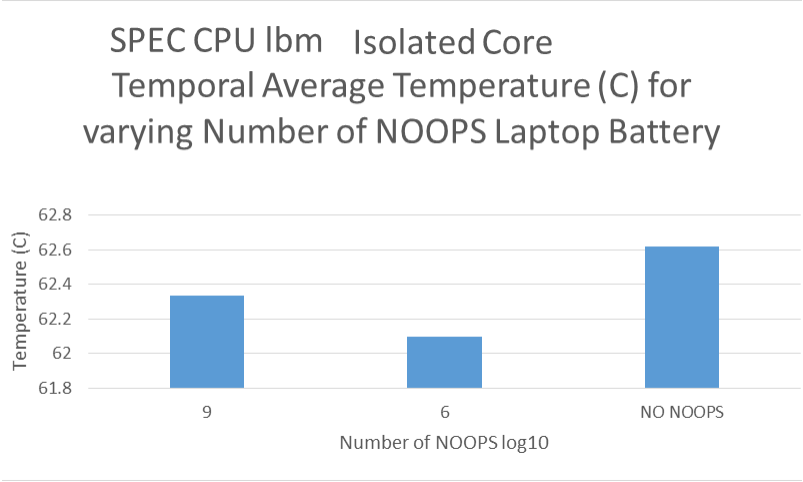

Figure 3.12: SPEC CPU lbm Measured Temporal Average Temperature for Isolated Core (C) for varying Number of NOOPS Laptop Battery - Linux

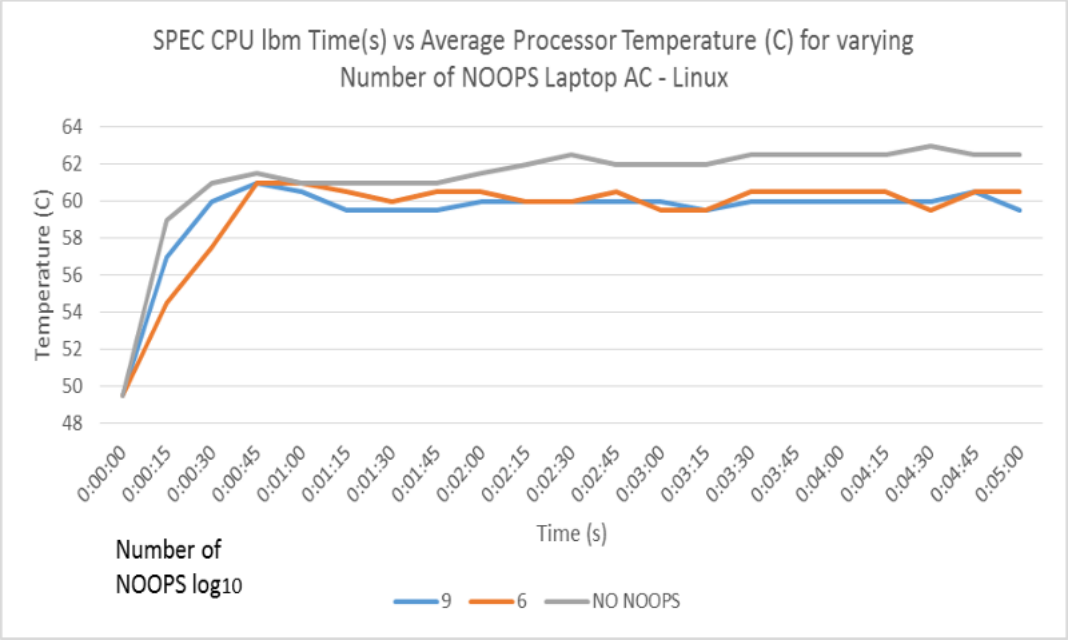

Figure 3.13: SPEC CPU lbm Time(s) vs Measured Spatial Average Temperature over All Cores (C) for varying Number of NOOPS Laptop AC - Linux

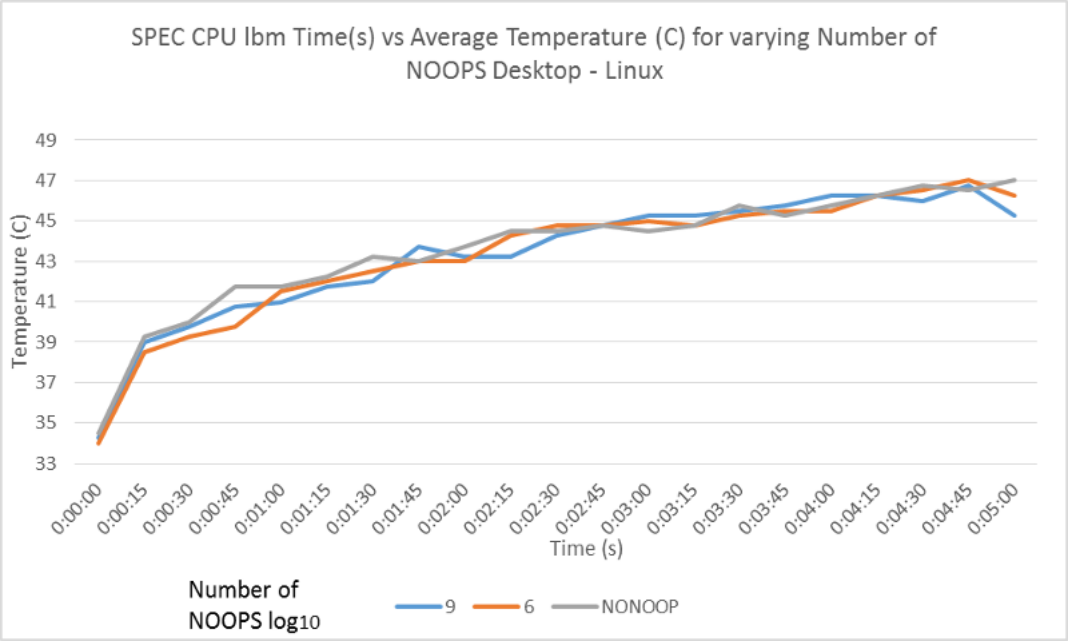

Figure 3.14: SPEC CPU lbm Time(s) vs Measured Spatial Average Temperature over All Cores (C) for varying Number of NOOPS Desktop - Linux

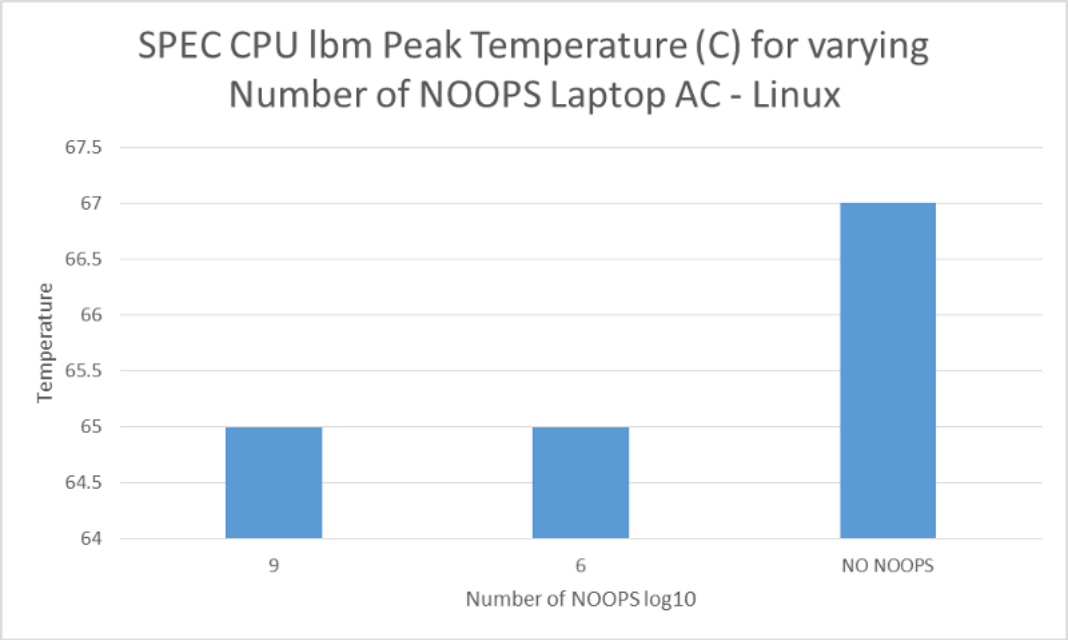

Figure 3.15: SPEC CPU Measured Temporal Peak Temperature (C) for varying Number of NOOPS Laptop AC - Linux

3.15 indicates differences in Peak Temperature over all cores for varying number of NOOPS for the Lattice Boltzman Method. This corresponds closely to the steady state temperature indicated in projections. Fig 3.16 indicates spatial and temporal average temperature differences in CPU temperature for the Lattice Boltzman Method on the Laptop with AC. Fig 3.17 and 3.18 provide spatial and temporal average temperature data and peak temperature data for the Lattice Boltzman Method in the Laptop AC and Desktop environments.

3.8 Shor's Factorization

Shor's factorization is a quantum algorithm for factoring integers. Quantum bits allow integers to be represented using fewer bits resulting in a lower time complexity. We insert NOOPS into a section of code which performs a matrix multiplication on a quantum register.

Fig 3.19 - Fig 3.20 indicate the differences in the average temperature for all cores for varying number of NOOPS for the Shor's factorization algorithm for Linux Laptop on AC and the Linux Desktop. Fig 3.21 indicates differences in Peak Temperature over all cores for varying number

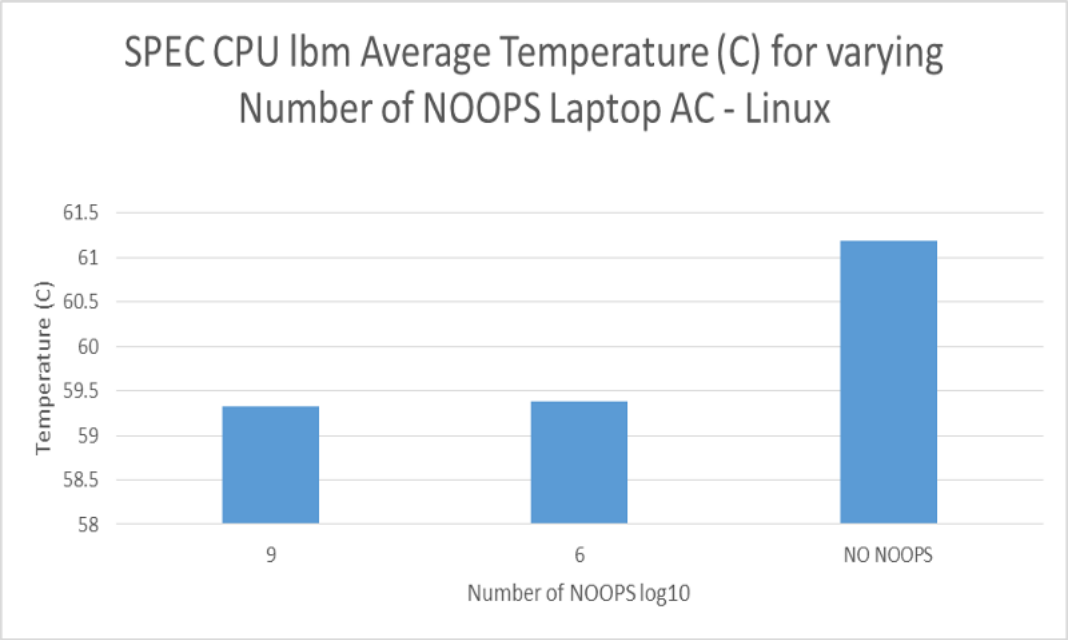

Figure 3.16: SPEC CPU Measured Average Temperature (C) for varying Number of NOOPS Laptop AC - Linux

of NOOPS for the Shor's factorization algorithm. This temperature reading corresponds closely to the steady state temperature indicated in projections. In the Shor's factorization simulation, it was determined that different sections of code were running which accounts for some of the difference in projected values. Fig 3.22 indicates average spatial and temporal average temperature differences in CPU temperature for the Lattice Boltzman Method.

Fig 3.23 and 3.24 provide spatial and temporal average temperature data and peak temperature data for the Lattice Boltzman Method.

3.9 sjeng chess simulation

sjeng is a chess simulation program. The program searches a tree to determine the best possible move for a player. We insert NOOPS into to a section of code used to search the tree of possible moves.

Fig 3.25 and 3.26 provides Isolated Core Temperature Data for low NOOP ranges for the sjeng simulation. Results indicate no differences in low NOOP ranges for the simulations in question. In

Number of NOOPS and Average Temperature across All Cores Laptop AC

Time	10E9	10E6	NO NOOPS
0:00:00	49.5	49.5	49.5
0:00:15	57	54.5	59
0:00:30	60	57.5	61
0:00:45	61	61	61.5
0:01:00	60.5	61	61
0:01:15	59.5	60.5	61
0:01:30	59.5	60	61
0:01:45	59.5	60.5	61
0:02:00	60	60.5	61.5
0:02:15	60	60	62
0:02:30	60	60	62.5
0:02:45	60	60.5	62
0:03:00	60	59.5	62
0:03:15	59.5	59.5	62
0:03:30	60	60.5	62.5
0:03:45	60	60.5	62.5
0:04:00	60	60.5	62.5
0:04:15	60	60.5	62.5
0:04:30	60	59.5	63
0:04:45	60.5	60.5	62.5
0:05:00	59.5	60.5	62.5
Average	59.33	59.38	61.19
Peak	65	65	67

Figure 3.17: lbm Number of NOOPS and Measured Average Temperature across All Cores Laptop AC

Number of NOOPS and Average Temperature across All Cores Desktop

Time	10E9	10E6	NONOOP
0:00:00	34.25	34	34.5
0:00:15	39	38.5	39.25
0:00:30	39.75	39.25	40
0:00:45	40.75	39.75	41.75
0:01:00	41	41.5	41.75
0:01:15	41.75	42	42.25
0:01:30	42	42.5	43.25
0:01:45	43.75	43	43
0:02:00	43.25	43	43.75
0:02:15	43.25	44.25	44.5
0:02:30	44.25	44.75	44.5
0:02:45	44.75	44.75	44.75
0:03:00	45.25	45	44.5
0:03:15	45.25	44.75	44.75
0:03:30	45.5	45.25	45.75
0:03:45	45.75	45.5	45.25
0:04:00	46.25	45.5	45.75
0:04:15	46.25	46.25	46.25
0:04:30	46	46.5	46.75
0:04:45	46.75	47	46.5
0:05:00	45.25	46.25	47
Average	43.7875	43.7625	44.0625
Peak	50	52	51

Figure 3.18: lbm Number of NOOPS and Measured Average Temperature across All Cores Desktop

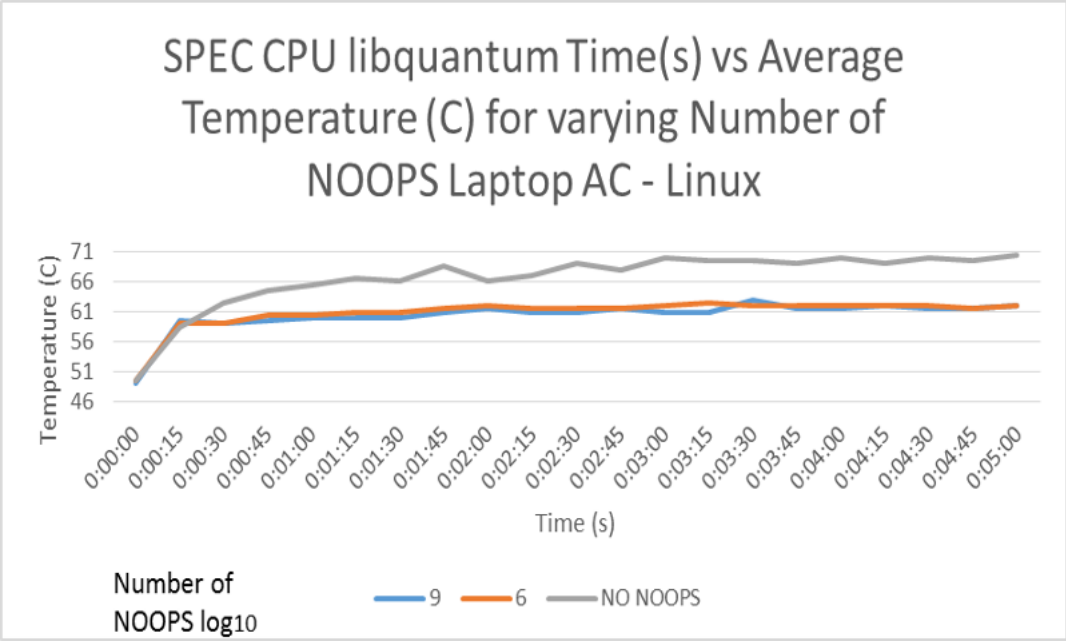

Figure 3.19: SPEC CPU libquantum Time(s) vs Measured Spatial Average Temperature over All Cores (C) for varying Number of NOOPS Laptop AC - Linux

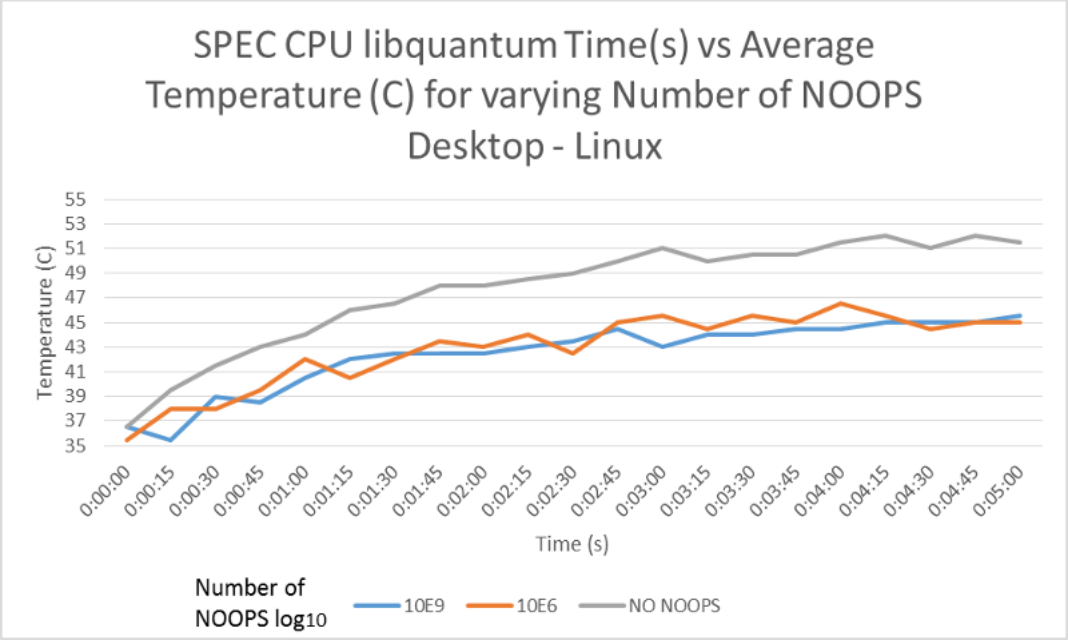

Figure 3.20: SPEC CPU libquantum Time(s) vs. Measured Spatial Average Temperature over All Cores (C) for varying Number of NOOPS Desktop - Linux

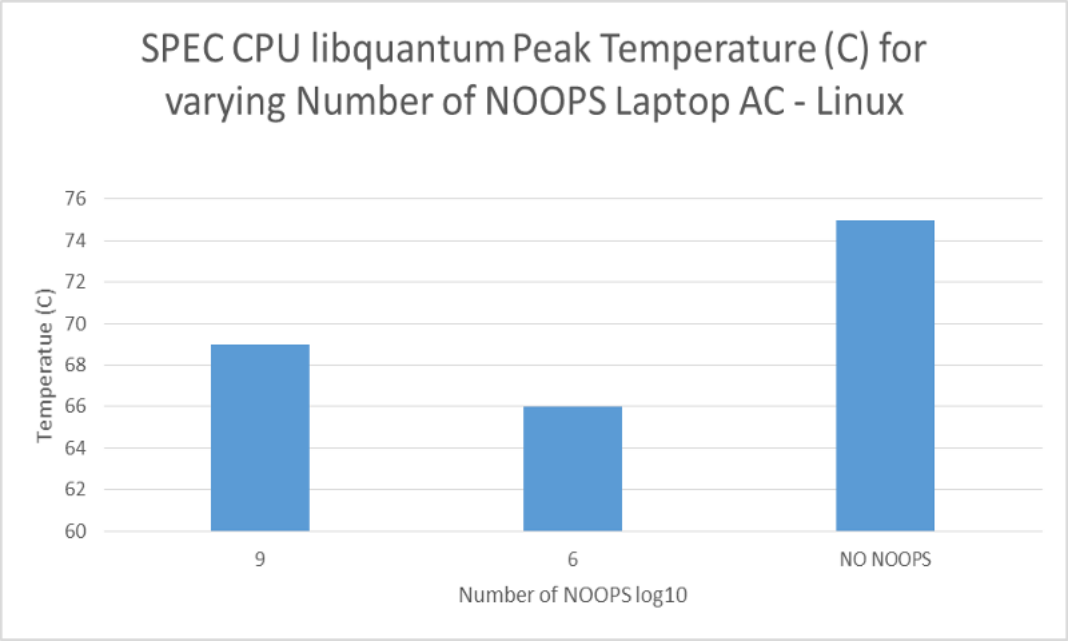

Figure 3.21: SPEC CPU libquantum - Measured Temporal Peak Temperature Across All Cores (C) for varying Number of NOOP : Laptop AC - Linux

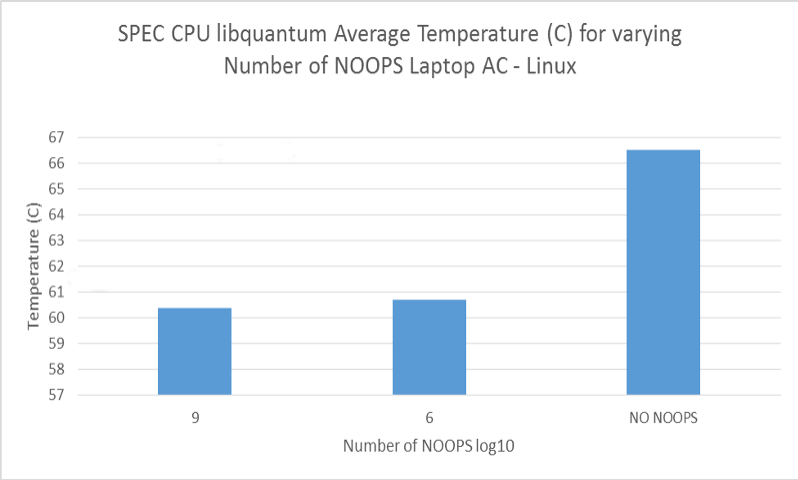

Figure 3.22: SPEC CPU libquantum Measured Average Temperature (C) for varying Number of NOOPS Laptop AC - Linux

1mincooldowntest10e9	20mincooldowntest10e9
43.84523C	41.40476C

Table 3.8: Cooldown Data Lattice Boltzman Method at 10e9 NOOPS

Fig 3.27 and 3.29, there is a observable difference in the srand simulation due to the small path size. Depending on the size of the path, low NOOP ranges may have an effect. If the path size is large, then a statistical test can be done to determine whether measurements of differences in temperature at low NOOP ranges are statistically significant. This was not done for these measurements, but it can be clearly observed that the probability that the measurement of a temperature difference in the sjeng measurement is statistically significant is less than 0.10 based on observed variance of the distribution. If the temperature is in fact statistically measured to be significant, it is because of the specific instruction values result in differences temperature readings and possibly also due to L2 cache misses and some of the the effects mentioned by Tiwari.

Fig 3.28 provides spatial and temporal average temperature data and peak temperature data for the sjeng simulation for on the Linux Desktop. Fig. 3.30 indicates the differences in the average peak temperature for all cores for varying number of NOOPS for the sjeng simulation for the Windows Desktop Environment.

Number of NOOPS and Average Temperature across All Cores – Laptop AC

Time	10E9	10E6	NO NOOPS
0:00:00	49	49.5	49.5
0:00:15	59.5	59	58.5
0:00:30	59	59	62.5
0:00:45	59.5	60.5	64.5
0:01:00	60	60.5	65.5
0:01:15	60	61	66.5
0:01:30	60	61	66
0:01:45	61	61.5	68.5
0:02:00	61.5	62	66
0:02:15	61	61.5	67
0:02:30	61	61.5	69
0:02:45	61.5	61.5	68
0:03:00	61	62	70
0:03:15	61	62.5	69.5
0:03:30	63	62	69.5
0:03:45	61.5	62	69
0:04:00	61.5	62	70
0:04:15	62	62	69
0:04:30	61.5	62	70
0:04:45	61.5	61.5	69.5
0:05:00	62	62	70.5
Average	60.381	60.7	66.5
Peak	69	66	75

Figure 3.23: libquantum Number of NOOPS and Measured Average Temperature Across All Cores Laptop AC

lbm10e6	lbm10e9	sjeng10e6	sjeng10e9	libquantum10e6	libquantum10e9
0.000788	8.27586e-7	”NO CHANGE”	0.9375	0.00306	0.00137

Table 3.9: Runtime Penalties For Lattice Boltzman Method, sjeng chess, and Shor’s Factorization

Number of NOOPS and Average Temperature Across All Cores Desktop

Time	10E9	10E6	NO NOOPS
0:00:00	36.5	35.5	36.5
0:00:15	35.5	38	39.5
0:00:30	39	38	41.5
0:00:45	38.5	39.5	43
0:01:00	40.5	42	44
0:01:15	42	40.5	46
0:01:30	42.5	42	46.5
0:01:45	42.5	43.5	48
0:02:00	42.5	43	48
0:02:15	43	44	48.5
0:02:30	43.5	42.5	49
0:02:45	44.5	45	50
0:03:00	43	45.5	51
0:03:15	44	44.5	50
0:03:30	44	45.5	50.5
0:03:45	44.5	45	50.5
0:04:00	44.5	46.5	51.5
0:04:15	45	45.5	52
0:04:30	45	44.5	51
0:04:45	45	45	52
0:05:00	45.5	45	51.5
Average	42.42	42.88	47.64
Peak	52	51	60

Figure 3.24: libquantum Number of NOOPS and Measured Average Temperature Across All Cores Desktop

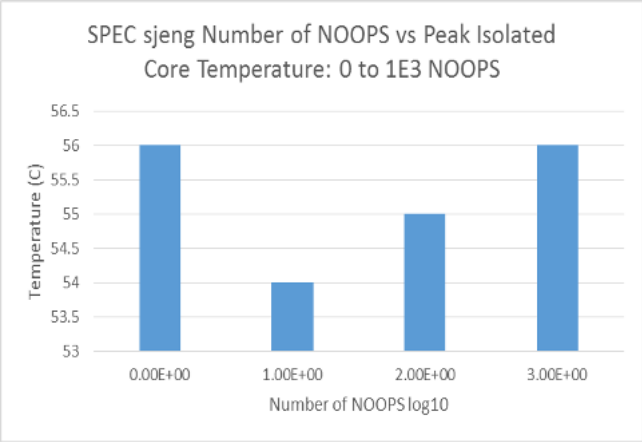

Figure 3.25: SPEC CPU sjeng Measured Peak Temperature on Isolated Cores (C) for varying Number of NOOPS (0 to 10E3) Laptop AC - Linux

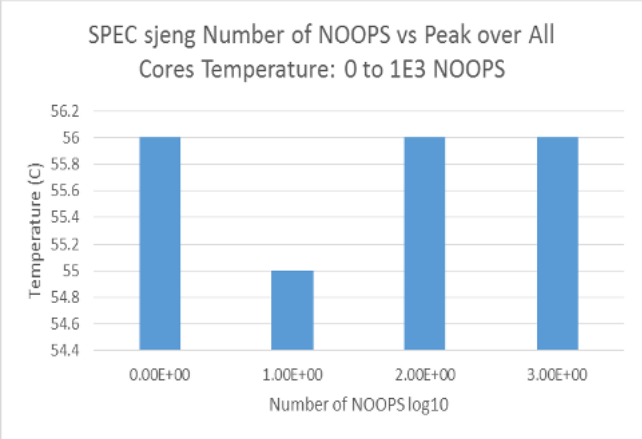

Figure 3.26: SPEC CPU sjeng Measured Peak Temperature over All Cores (C) for varying Number of NOOPS Laptop AC (0 to 10E3) - Linux

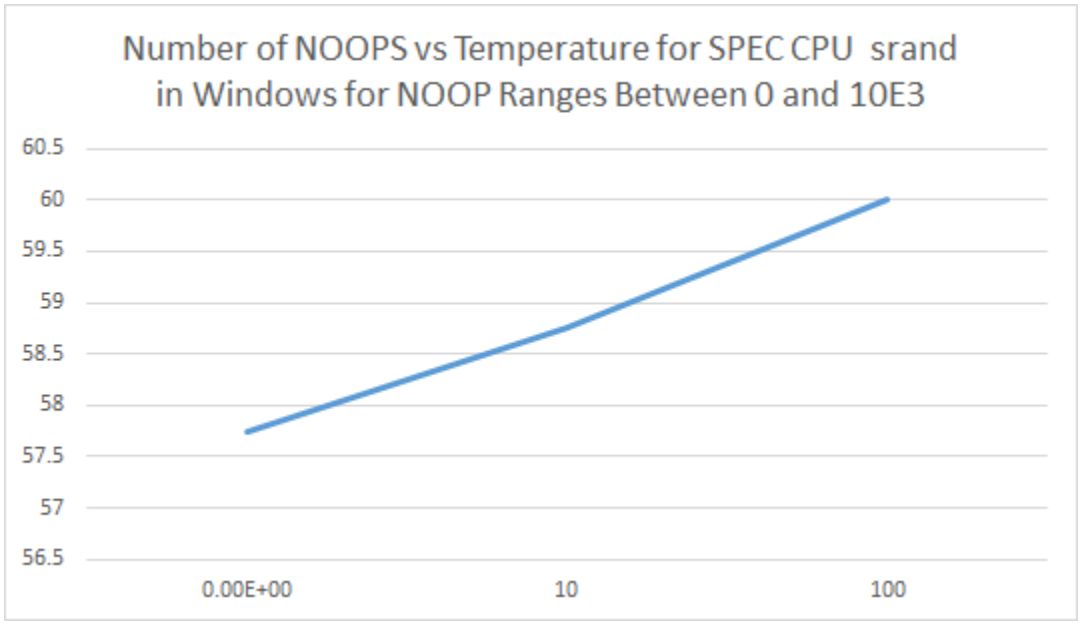

Figure 3.27: Number of NOOPS vs Temperature for SPEC CPU srand in Windows for NOOP Ranges Between 0 and 10E3

Number of NOOPS And Average Temperature Across All Cores (Desktop Averages Adjusted to Same Starting Temp)

Time	10E9	10E6	NO NOOPS
0:00:00	33.5	34.5	35.25
0:00:15	39.25	40.5	41.75
0:00:30	41.5	42.5	43
0:00:45	42.5	44.25	44.5
0:01:00	44.25	44.75	45.5
0:01:15	45	45.5	46
0:01:30	45.5	46.5	47.25
0:01:45	46.5	48	48.25
0:02:00	46.75	47.75	47.75
0:02:15	48.25	48.5	48.75
0:02:30	49	49.25	49.25
0:02:45	49	49.5	50
0:03:00	49.75	50.75	49.75
0:03:15	49.5	50.75	50.75
0:03:30	50.5	50.5	51.25
0:03:45	50.5	51	51
0:04:00	50.5	51	51.75
0:04:15	52	51.25	52.5
0:04:30	51	51.25	52
0:04:45	52.5	51.75	52.75
0:05:00	52.25	52.75	52.25
Average	**50.59091**	**50.88636**	**51.20455**
Peak	**59**	**58**	**60**

Figure 3.28: sjeng Measured Number of NOOPS And Average Temperature Across All Cores (Linux Desktop)

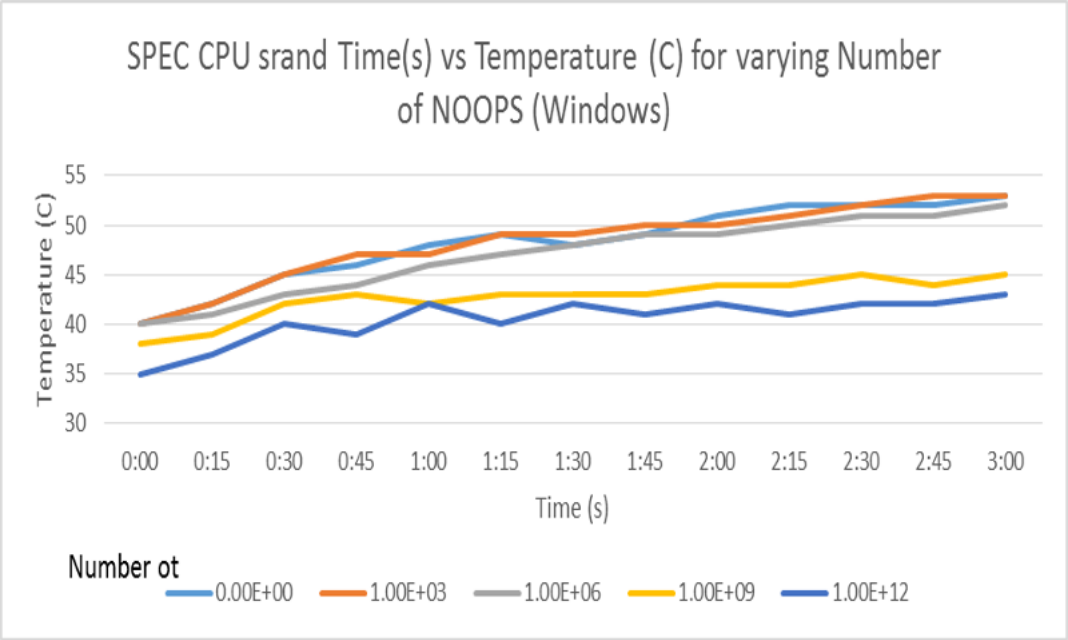

Figure 3.29: srand Time(s) vs Measured Average Core Temperature for varying Number of NOOPs

The primary hypothesis is that memory instructions are the primary drivers of energy consumption which is indicated by previous work [1], and this hypothesis is used to create a novel process model. In order to verify this model, it is necessary to verify the power usage from memory instructions from a model of the processor.

Prior to doing this, some preliminary work is done by simply counting each instruction in a process model. It is not sufficient to only count the instructions because attributes associated with each instruction are also used to determine the power dissipation. The estimated power reduction for a 100 MHZ processor is described above. When NOOPS were inserted, temperature reductions were also in the range predicted by the NOOP ratios unless different sections of code were running. The Windows based average peak chart does not use the same calculation methodology as the other measurements of peak temperature because sensor readings were not taken using a programmatic sensor reading. This is the reason a different calculation methodology was used. The measurements of peak temperature are otherwise consistent with predictions.

It was mentioned earlier that the full hardware logic associated with an implementation was not specified. The SystemC sample code contains an example of a RISC CPU design that we

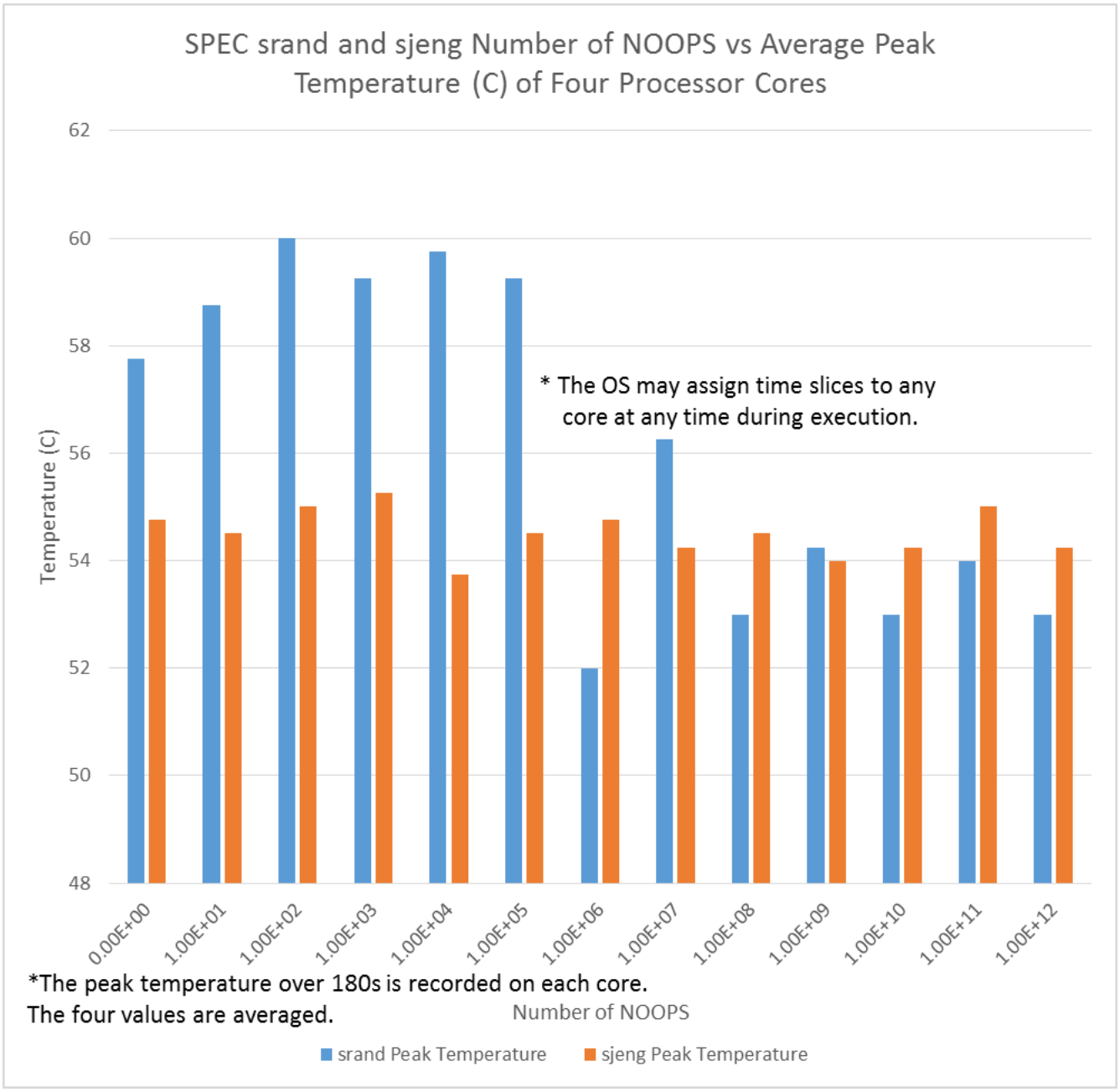

Figure 3.30: SPEC srand and sjeng Average Peak Temperature (C) over Four Processor Cores (Windows) (The peak temperature at a NOOP range is recorded on each processor core. These values are averaged.)

would have to modify if one were to actually do a hardware design. Instead of modifying the RISC CPU design to work with our simulation code, a FIFO scheduler was modified to work as a round robin based process scheduler. Assembly sequences are read in this simulation and NOOPS are assigned to processes on the basis of the algorithms coded in Java.

3.10 Analysis of Results

There is always a temperature reduction when NO NOOPS are used and when 10e9 NOOPS are used. The difference temperature between 10e6 and 10e9 is generally much smaller or statistically nonexistent due to the fact the NOOP ratio may not necessarily not change very much in this range. This has to do with how the NOOP ratio changes in the critical loop/path of the program as NOOP loop iterations are increased in software. The NOOP ratio is believed to be in the range 0.6 when the runtime penalty is high in the simulations we did. In our previous simulation work, we attempt to make projections of the temperature reduction on this ratio and our results are consistent except in the case of the libquantum simulation in which case the slowdown is so severe that different sections of code are running during the simulation.

There is a natural variation in the average temperature over a simulation. However, since variables being averaged are not assumed to be independent random variables, we can not assume that mean will be normally distributed without additional assumptions. (We do know that the mean of simulation set averages is normally distributed.) If we assume that each measurement (average core temp at time t) follows the same distribution (which we do not know) with some linear transformation applied to it over the course of the simulation, for instance, +.2 C, then we can then assume that each variable is an independent random variable, and that the mean of the simulation set will be normally distributed regardless of the underlying distribution. It is thus established using the central limit theorem that the average processor temperature measurements are normally distributed.

In the Linux (monolithic kernel) environment, the kernel runs on all cores. In Windows, various portions of the NT kernel are assigned various cores at various times according to a scheduling

policy. In both cases, the portions of the kernel run on any core that has any processes running on it.

The processor assigns time slices to processor cores according to a scheduling policy. Whether a particular process runs on a particular core is determined by this policy. There is no trivial way to determine how the OS assigns time slices at runtime, but in Linux and Windows the programmer may assign an affinity for a particular processor. This is done in Linux simulations.

The ambient temperature was 73F Desktop Windows simulations, 75F on Desktop Linux simulations and 73F on Laptop Linux simulations. The cool down test performed in a Linux environment.

The desktop machine had the following specifications: Memory 8.00 GB Ram, 240 GB disk, Model: Dell Optiplex 7020, Processor: Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz, 4 Core(s), 8 Logical Processor(s), L2 cache: 1024kb, L3 cache: 8192kb.

The laptop machine had the following specifications 4.00 GB RAM, 156 GB Disk, Latitude E3250, Intel Core i5 2520 2.50 GHZ, 4 logical processors. L2 Cache: 256kb, L3 Cache: 3072kb.

3.11 Conclusion

A process buffer is simulated with NOOP insertions to hold a sequence of instructions in a process set before they are executed. Previous work to describes the physical architecture of the buffer and its relation to the processor. In this extension, the buffer holds a larger series of nested loops which encompasses the critical sections of a set of processes. The SystemC simulation language is used to verify that this can be implemented in hardware, though the design is not fully implemented. Better specifying the implementation is an area of further research.

A disassembler is used to determine a possible execution path. For this work, the IDA pro to disassembler is used to disassemble Apache executable. Critical path(s) are reconstructed in hardware and NOOPS inserted for validation purposes. Future research may disassemble programs on the ARM architecture as opposed to x86. A counter is used to count the total number of instructions in a critical sections and also the number of memory references. Instructions are

written to the buffer if the ratio of these two numbers is less than a predetermined ratio. If it is greater, we write a NOOPS to the buffer until it is below this number, and then write the remaining instructions. Temperature reduction projections are made on the basis of the above simulations and compared to actual results. The work for this project is simulation work and we can do further research along this line. The approach described is to take critical sections of processes and insert NOOPS resulting in a lower power dissipation. Future research may attempt to determine the minimum slowdown that is required before no additional temperature reductions are observed.

Bibliography

[1] V. Tiwari, S. Malik, A. Wolfe, M. Lee, Instruction Level Power Analysis, Journal of VLSI Signal Processing Systems, no.1, pp.2232383, 1996

[2] T. Simunic, L. Benini, G. De Micheli and M. Hans, Source Code Optimization and Profiling of Energy Consumption in EmbeddeSystems, Proceedings of the 13th International Symposium on System Synthesis, pp 193-198, September 2000.

[3] H. Tomyiama, H., T. Ishihara, A. Inoue, H. Yasuura, Instruction scheduling for power reduction in processor-based system design, DATE, 1998.M. Young, The Technical Writers Handbook. Mill Valley, CA: University Science, 1989.

[4] N. Beckmann and D. Sanchez. Jigsaw: Scalable Software-Defined Caches. In Proc. PACT-22, 2013

[5] M Knoth US Patent 7,873,820 - Processor utilizing a loop buffer to reduce power consumption, 2011

[6] Intel Corporation, ”2nd Generation Intel Core Processor Family Desktop and Intel Pentium Processor Family Desktop, and LGA 1155 Socket Thermal Mechanical Specifications and Design Guidelines”