Analyzing Compute-Intensive Software Performance
by
Neeraj Gupta
Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of
Bachelor of Science
and
Master of Science
in Computer Science and Engineering
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
May 1995
© Neeraj Gupta, MCMXCV. All rights reserved.
The author hereby grants to MIT permission to reproduce and distribute publicly
paper and electronic copies of this thesis document in whole or in part, and to grant
others the right to do so.

Author..............................................................................................................
Department of Electrical Engineering and Computer Science
May 12, 1995

Certified by ...........................................................................................................
John V. Guttag
Professor, Associate Department Head Computer Science & Engineering, MIT
Thesis Supervisor (Academic)

Certified by ...........................................................................................................
William D. Evans
Principle Software Engineer, Xerox Corp. (XSoft)
Thesis Supervisor (Cooperating Company)

Accepted by...........................................................................................................
Frederic R. Morgenthaler
Chairman, Committee on Graduate Students
JUL 17 1995
Analyzing Compute-Intensive Software Performance
by
Neeraj Gupta

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 1995, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science
and
Master of Science
in Computer Science and Engineering

Abstract
Understanding the hardware utilization of a software system can aid in improving its performance. This thesis presents a General Outline for Operational Profiling (GOOP) approach to modeling hardware utilization by an instruction stream. The GOOP approach provides an inexpensive and portable method to determine a software system's instruction-level performance characteristics. An important aspect of this approach is that it requires no hardware instrumentation. We use the GOOP method to mathematically model the system utilization of an existing software-only virtual machine emulator and apply the model to guide modifications that improve the emulator's overall performance.

Thesis Supervisor (Academic): John V. Guttag
Title: Professor, Associate Department Head Computer Science & Engineering, MIT

Thesis Supervisor (Cooperating Company): William D. Evans
Title: Principle Software Engineer, Xerox Corp. (XSoft)
Acknowledgements

We, actually I (there is no we), would like to acknowledge everybody who helped me stay sane during this entire thesis creation process. However, a few individuals need to be thanked personally. A very special thanks to Ken; I didn’t know him, and he didn’t know me, but his brilliant insight into common sense was the spark that enabled this research. Thanks to Bill for his ideas, suggestions, and laughs—and for being there to listen to my crazy ideas about how to make the world faster. Thanks Max, the acronym master, for bleeding over countless revisions of this document. Mr. Flippin was just plain important. And a closing thanks to the people who wrote Maple, the program that solved the GOOP system utilization model used as the basis for this work.
Contents

1 Introduction ................................................. 13
  1.1 Background ................................................... 13
    1.1.1 Predictive Modeling of Software Performance ............. 14
    1.1.2 Descriptive Modeling of Software Performance ............ 14
  1.2 GOOP Approach to Modeling Hardware Utilization .......... 15

2 Developing a System Utilization Model ....................... 17
  2.1 Memory Subsystem .......................................... 17
    2.1.1 Components ............................................. 17
    2.1.2 Dynamics ................................................ 18
  2.2 General Descriptive Model ................................... 18
    2.2.1 L1 and L2 Cache States ................................ 18
    2.2.2 Execution Time with Caches (no Overlap) ............... 20
    2.2.3 Accounting for Overlap .................................. 20
    2.2.4 Relation to Number of Memory Operations ............... 21
  2.3 System Utilization Model ................................... 22
    2.3.1 Underlying Principle .................................... 22
    2.3.2 Simple Example .......................................... 23
    2.3.3 Application to Test Platform ............................ 23

3 Validation with Consistency Checks ............................ 27
  3.1 Test Properties ............................................. 27
  3.2 Test 1 .................................................... 28
  3.3 Test 2 .................................................... 28
  3.4 Test 3 .................................................... 30
  3.5 Test Limitations ............................................ 32

4 Application to Software System ............................... 33
  4.1 Background on the GVWin System ............................ 33
  4.2 Applying the System Utilization Model ...................... 34
  4.3 Performance of Baseline GVWin ............................. 35
  4.4 Performance of Modified GVWin ............................. 36

5 Conclusions .................................................. 39
  5.1 Summary .................................................... 39
  5.2 Evaluation .................................................. 39
  5.3 Future Work ............................................... 40
List of Figures

3-1  Pseudocode for Test 1 ........................................ 28
3-2  Pseudocode for Test 2 ........................................ 29
3-3  Pseudocode for Test 3 ........................................ 31
4-1  Internal timers implemented in Guam ......................... 35
List of Tables

3.1 Comparison of Predicted vs. Model outputs for Test 1 ....................... 28
3.2 System utilization for Test 1 ................................................. 29
3.3 Comparison of Predicted vs. Model outputs for Test 2 ..................... 29
3.4 System utilization for Test 2 ............................................... 30
3.5 Comparison of Predicted vs. Model outputs for Test 3 ..................... 30
3.6 System utilization for Test 3 ............................................... 31

4.1 Performance profile for baseline Midway (BOOT) ......................... 35
4.2 Performance profile for baseline Midway (PAG) .......................... 36
4.3 Cache hit rates for baseline Midway ........................................ 36
4.4 Performance profile for modified Midway (BOOT) ....................... 37
4.5 Performance profile for modified Midway (PAG) .......................... 37
4.6 Cache hit rates for modified Midway ........................................ 37
4.7 Compare Baseline v. Modified Midway Performance (BOOT) .......... 38
4.8 Compare Baseline v. Modified Midway Performance (PAG) .......... 38
Chapter 1

Introduction

Modern computer architectures include complex processing elements with complex memory systems that introduce difficult to quantify effects on the speed of instruction execution. This hardware complexity can make it difficult to predict and fully understand the performance characteristics of software systems; as a result, software systems frequently do not achieve the level of performance predicted by analyzing the source code and dynamic behavior of the software. Often, the discrepancy between expected and actual performance is machine-dependent and can be attributed to the underlying hardware complexity [10].

Determining the hardware utilization of such a software system can aid in improving its performance [10]. This thesis develops a General Outline for Operational Profiling (GOOP) approach to modeling hardware utilization. It consists of measuring execution time for a test run of the software system, then modifying the underlying hardware slightly and measuring execution time of the same test run (on the new hardware configuration). Repeat this process with carefully planned hardware modifications. Finally, infer hardware utilization by analyzing the differences in execution times between the trials.

We obtain a hardware utilization profile for a software-only virtual machine emulator (GVWin, see Section 4.1) by applying the GOOP modeling technique outlined above. Based on the hardware utilization profile, we determine GVWin’s instruction-level performance and accordingly modify the emulator to improve its overall performance.

Chapter 1 provides information regarding the applicability of mathematical models in improving software performance and outlines the GOOP approach to modeling hardware utilization. Chapter 2 uses the GOOP approach to build a model suitable for determining the hardware utilization of some instruction stream executing on a test platform. Chapter 3 validates the model by checking the model outputs against known values. Chapter 4 illustrates usage of the model in order to guide modifications to improve the performance of GVWin. Chapter 5 discusses some relevant conclusions based on this thesis.

1.1 Background

Mathematical models can provide a convenient and conceptually easy way to understand how time is spent during execution of a software system, i.e. a way to understand that system’s performance behavior. Creation of a dynamic performance profile based on models can guide performance enhancing modifications to the system. In general, two types of models are constructed [4]:
Predictive Models attempt to predict the performance of the software system. A predictive model constructed prior to system development can be used as a design tool to aid in creating a system that meets predefined performance criteria. Varying the input parameters to the model should show how the related performance profile for the system changes. See Section 1.1.1.

Descriptive Models attempt to describe the actual performance of the software system. Construction of a descriptive model after creation of the software system allows determination the actual dynamic performance profile. These models may mirror a previously constructed predictive model. See Section 1.1.2.

Both predictive and descriptive models can be based on a more general execution model [5]:

1. Partition the overall execution of the software system into a set of events such that the union of these events represents the entire execution. At the highest level, one event that corresponds to the entire program is sufficient.

2. Gather information regarding the frequency and distribution of each event.

3. Gather information regarding the time to process each event.

4. Create a mathematical model representing how time is spent during execution of the program.

Analysis of the execution model describes, at the granularity of any event, the performance profile for the software system under consideration.

1.1.1 Predictive Modeling of Software Performance

Predicting the time to process each event in some instruction stream (by examining the instructions executed, algorithms, usage patterns, etc.) allows construction of a predictive model based on the execution model for that instruction stream [2, 4].

For software systems that are analyzed at the instruction level, the predictive model rarely corresponds to the observed performance because modern computer architectures include complex processing elements with complex memory systems [10] (for example, the Intel 486 processor has an on chip instruction/data cache, instruction prefetch buffers, and uses simple code reordering algorithms). The static behavior of system sub-elements within the simplified processor–memory abstraction can be defined precisely, but the actual behavior of these sub-elements is dynamic and their performance varies with each unique instruction sequence [7, 6]. Directly predicting the instruction-level performance characteristics of compute-intensive software systems (utilization of primarily the processor and memory subsystems, as opposed to I/O, constitutes a compute-intensive software system) becomes difficult because of the high level of interdependence between processor and memory sub-elements and the dynamic nature of the instruction stream [10].

1.1.2 Descriptive Modeling of Software Performance

Predictive modeling is not the only way to gain an understanding of compute-intensive software systems. An analytic descriptive model based on the execution model (see Section 1.1)
can be constructed if the system already exists and the time to process each event of the
execution model can be determined [4].

At a high level, each event under consideration is a software event and can easily be
measured directly (using some type of clock) [5]. At the instruction level, however, events
under consideration are hardware-based (processor execution times, cache service, memory,
buses, etc.). In order to construct accurate instruction-level models, it is necessary to
quantify effects that the complexity of the underlying hardware architecture imposes upon
instruction execution. Standard methods for determining these effects on actual systems
use hardware-based measurement tools [10]:

1. Using hardware probes and monitors attached to accessible buses and chip contacts,
gather detailed trace information regarding the state of the sub-element being studied
(for all time in the test). Note that this becomes increasingly difficult as processor
complexity increases. For example, the Intel 486 processor has an on-chip 8kB instruc-
tion/data cache that is not accessible by off-chip probes. To gather data regarding
the on-chip cache, one must use a specialized 486 logic analyzer that simulates 486
internal cache behavior.

2. Consolidate data to provide meaningful results that detail hardware sub-element util-
ization.

Hardware-based measurement tools are effective and accurate, but normally provide
much more detailed information than is needed in order to gain an understanding of how
hardware utilization affects low-level performance characteristics of the software (for exam-
ple, information about specific pin and bus line usage can be obtained for each machine
cycle). The tools required are generally not portable across platforms, and the cost in terms
of time, knowledge, and money, needed to use hardware-based measurement tools is high
compared with software analysis techniques that give more general information regarding
high-level performance of the software system.

1.2 GOOP Approach to Modeling Hardware Utilization

We now present a novel approach to modeling utilization of hardware sub-elements for
instruction-level analysis of compute-intensive software that does not suffer from the draw-
backs of conventional hardware-based techniques.

The GOOP approach allows a hardware utilization profile to be determined by construct-
ing a mathematical model of the hardware platform, then solving the model with respect
to timing data gathered for runs of the software system on carefully modified versions of
the hardware platform (that correspond to the model).

1. Create a general descriptive model for the system platform (analogous to the execution
model used in Section 1.1.2 and described in Section 1.1). The events used should
represent time spent in the hardware sub-elements under consideration.

2. Physically modify one of the execution parameters of the given system and create an
updated general descriptive model based on this change. The two models should have
similar forms including many like terms with known relationships (to aid in Step 3).

3. Link the two general descriptive models to create a system utilization model. Because
of the similarity in the models created in Step 2, terms can be substituted and sub-
tracted in order to reduce the number of dependent variables in the resulting system
of equations. Since execution time is directly and easily measured, consider time an independent variable.

4. Repeat from Step 2 until the resulting system of equations (the system utilization model) can be solved. If the resulting model cannot eventually be solved (at least numerically), it may be necessary to instantiate parts of the model (some dependent variables) with known, accepted, or predicted values.

5. Solve the system utilization model. This describes the performance of the system with respect to the hardware sub-elements that were used in step 1.

Once the system utilization model has been created for some hardware platform, it can be applied to any software system running on that platform. In this manner, the utilization of the underlying hardware sub-elements can be quantified (for some instruction stream) and instruction-level performance can be analyzed.

For example, assume a system that spends time either in the processor (executing instructions) or in main memory (servicing memory requests). Section 2.3.2 describes how the GOOP approach can determine, based on only execution times of an instruction stream, how the total execution time was partitioned between time in the processor and time in main memory for that instruction stream. This data clearly provides information regarding instruction-level performance and can identify hardware-based performance bottlenecks that would otherwise not be easily seen. Optimization for the given software system can now take into account the underlying hardware complexity.
Chapter 2

Developing a System Utilization Model

This section describes the creation of a system model suitable for determining hardware utilization by some fixed instruction stream. We use an IBM ValuePoint 433D computer (using a 486D-class microprocessor) throughout as the test platform.

Section 2.1 discusses the memory subsystem of the test platform. Section 2.2 describes the creation of a general descriptive model of the test platform. Section 2.3 builds upon this general descriptive model to create the system utilization model. Note that for the following sections, hardware sub-element service times are given in terms of cycles when the service time is dependent on the speed of the system/processor clock, and are given in terms of seconds when the service time is absolute (not dependent on the speed of the system/processor clock).

2.1 Memory Subsystem

Understanding of the memory subsystem is an integral part of building the system utilization model. The memory subsystem for the test machine can be decomposed into the actual components and their dynamic behavior when servicing processor-initiated memory requests (DMA requests are not considered here as they have no effect on the caches or the processor-memory bus).

2.1.1 Components

The memory subsystem of the test platform consists of the following components:

**L1 Cache** 486-class microprocessors contain an on-chip 8 kB shared instruction/data cache. This cache is 4-way set associative and uses an LRU replacement policy.

**L2 Cache** The standard 433D motherboard has a direct-mapped Level 2 cache (128 kB) placed serially before main memory and is clocked by the system (motherboard) clock. The 15ns SRAM used allows cache response time of approximately 20ns.

**Main Memory** 70ns DRAM used allows main memory response time in the range of 107ns-121ns.
2.1.2 Dynamics

486 processor instructions that use memory assume L1 cache hits in the normal case. Processing proceeds in parallel with L1 cache lookups, so hitting in the L1 cache does not add to the execution time of any instruction. In the event of a miss in the L1 cache, memory requests are handled in the following manner:

- The processor presents the memory request to the off-chip memory subsystem and immediately stalls for 2 processor cycles (the L1 miss penalty).
- If the L2 cache hits, it returns data to the processor immediately. The processor continues normal execution when \( \max(\text{L1 miss penalty}, \text{L2 service time}) \) has expired. The request is not presented to main memory.
- If the L2 cache misses, it presents the request to main memory. There is no additional miss penalty for the L2 cache, i.e. a miss in the L2 cache takes the same amount of time as a hit, the L2 service time. Since the main memory response time is much greater than the L1 miss penalty, the processor always continues when the data becomes available. L2 data replacement proceeds in parallel with returning data to the processor.

2.2 General Descriptive Model

A general descriptive model of the test platform must be built before creating a system utilization model. It is important that the general descriptive model be complete enough to be sufficiently accurate, but be simple enough to be easily understood and used. We develop equations for the test platform that:

- describe how, during program execution, time is allocated in the hardware for differing states of the L1 and L2 caches
- account for overlap between processor execution and memory response
- relate subsystem time to the number of memory operations

2.2.1 L1 and L2 Cache States

During program execution, we partition overall execution time \((\text{Time})\) of an instruction stream into time spent actually processing instructions \((P)\) and time spent due to the memory subsystem \((\text{Mem})\):

\[
\text{Time} = P + \text{Mem}
\]  
(2.1)

(This partitioning is a bit simplistic, since the test platform has some overlap between processor execution and memory response. This is dealt with in Section 2.2.3.)

As the L1 and L2 caches are systematically enabled and disabled, \(P\) remains constant while \(\text{Mem}\) changes in accordance with the effects of the L1 and L2 caches. Equations 2.2–2.4 describe how \(\text{Mem}\) changes with the 3 possible states of the L1 and L2 caches.\(^1\)

\(^1\)The L2 cache cannot be enabled if the L1 cache is disabled. Note that this situation is not particularly interesting in any case.
L1 and L2 Caches Disabled

Disabling the L1 cache in the test platform bypasses the on-chip cache mechanism. The processor presents memory requests to the external memory subsystem directly, incurring no L1 cache miss penalty. Disabling the L2 cache in the test platform causes all data requests for the L2 cache to miss (instead of bypassing the L2 cache mechanism completely)\(^2\) and be serviced by main memory. We describe time spent due to the memory subsystem under these conditions \((\text{Mem}''')\) by:

\[
\text{Mem}''' = \text{Main Memory Time} = C_{L2} + M
\]

where \(C_{L2}\) is time due to the L2 cache (servicing misses), and \(M\) is the time spent actually servicing requests in (or waiting on) main memory.

Only L1 Cache Enabled

Enabling the L1 cache in the test platform allows the processor to handle some of the memory requests internally; missing in the L1 cache causes the processor to stall because of the L1 cache miss penalty, but this time is masked by the much longer latency of main memory servicing the memory request. We describe time spent due to the memory subsystem under these conditions \((\text{Mem}')\) directly from Equation 2.2:

\[
\text{Mem}' = \text{Main Memory Time} = (1 - h_{L1})(C_{L2} + M)
\]

where \(h_{L1}\) is the hit rate of the L1 cache.

L1 and L2 Cache Enabled

Enabling the L2 cache in the test platform allows the L2 cache to handle some of the off-chip memory requests instead of presenting them to main memory. In the case where the L2 cache also misses, we do not consider the L1 cache miss penalty since this time is masked by the much longer latency of main memory response. We describe time spent due to the memory subsystem under these conditions \((\text{Mem})\) directly from Equation 2.3 and Section 2.1.2:

\[
\text{Mem} = \text{(L2 Hit Time) + (Main Memory Time)}
\]

\[
\text{Mem} = h_{L2}(\max(C_{L1}, C_{L2}) + (1 - h_{L1} - h_{L2})(C_{L2} + M))
\]

where \(h_{L2}\) is the hit rate of the L2 cache (in terms of total memory operations) and \(C_{L1}\) is the time due to the L1 cache miss penalty.

\(^2\)When the IBM 433D motherboard’s L2 cache is disabled, the WRTE (Write Enable) line to the L2 cache is enabled low. This keeps any valid data from ever getting written causing all L2 cache requests to miss.
2.2.2 Execution Time with Caches (no Overlap)

We write equations describing the total execution time for an instruction stream for when both the L1 and L2 caches are disabled ($T''$), when only the L1 cache is enabled ($T'$), and when both the L1 and L2 caches are enabled ($T$) (assuming no overlap) as:

\[ T'' = P + C_{L2} + M \]  \hspace{1cm} (2.5)

\[ T' = P + (1 - h_{L1})(C_{L2} + M) \]  \hspace{1cm} (2.6)

\[ T = P + h_{L2}(\max(C_{L1}, C_{L2})) + (1 - h_{L1} - h_{L2})(C_{L2} + M) \]  \hspace{1cm} (2.7)

2.2.3 Accounting for Overlap

Sections 2.2.1–2.2.2 assumed no overlap between processor execution and memory response. In this section, we extend the equations described in Section 2.2.2 to deal with overlapped execution.

When the L1 cache misses, the 486 processor presents the memory request to the off-chip memory subsystem. The processor immediately stalls for 2 cycles as the L1 miss penalty. If the data is available after the stall, the processor continues normal operation. If the data is still unavailable, the processor attempts to overlap instruction execution during the external memory subsystem response time. The simple algorithm for determining overlap execution used by the 486 is:

1. Let $I_m$ represent the instruction that caused the external memory request to be generated.

2. If the data for $I_m$ becomes available, complete $I_m$ and continue normal execution.

3. Execute subsequent instructions as long as they are available and legal; otherwise, stall until $I_m$ completes. Given that:
   - an available instruction is stored in the decode or prefetch buffer
   - a legal instruction does not access memory or use the destination register of $I_m$

We make the simplifying assumption that overlapped instructions can occur only when main memory services processor-initiated external memory requests.\(^3\)

Since $P$ is the amount of time spent in the processor (actually processing), Equation 2.5 should be modified:

\[ T'' = P + C_{L2} + M - O \]  \hspace{1cm} (2.8)

where $O$ is the time spent in overlapped processor execution and memory response.

When the L1 cache is enabled, it activates the 486 instruction prefetch unit (which stores sequential bytes from the instruction stream in a 128 bit buffer). This allows for a higher potential of legal overlap instructions, which can be significantly greater than with the L1 cache and prefetch unit disabled. We modify equations 2.6 and 2.7:

\[ T' = P + (1 - h_{L1})(C_{L2} + M - O') \]  \hspace{1cm} (2.9)

\(^3\)In reality, overlap can occur in the time $C_{L2} - C_{L1}$ (if any) as well. This situation is not considered for simplicity (see Section 2.3.1).
\[ T = P + h_{L2}(\max(C_{L1}, C_{L2})) + (1 - h_{L1} - h_{L2})(C_{L2} + M - O') \]  

(2.10)

where \( O' \) is the time spent in overlapped processor execution and memory response with the processor prefetch unit active.

Equation 2.10 describes the normal case execution time breakdown for programs executing on the test platform:

- \( T \): Total execution time
- \( P \): Time spent executing instructions in the processor (this includes L1 cache hits)
- \( h_{L2}(\max(C_{L1}, C_{L2})) \): Time spent for memory requests that hit in the L2 cache (the processor must stall for at least \( C_{L1} \) time before continuing, however)
- \( (1 - h_{L1} - h_{L2})(C_{L2} + M) \): Time spent for memory requests that are serviced by main memory (since the L2 cache in in series with main memory, \( C_{L2} \) must be included)
- \( (1 - h_{L1} - h_{L2})O' \): Time spent executing overlapped instructions (this is subtracted so overlap instruction time is not double-counted in \( P \) and \( M \))

### 2.2.4 Relation to Number of Memory Operations

For the following discussion, let \( M_1 \) represent the amount of time it would take for all processor initiated memory requests to be serviced if each request could be handled in exactly 1 processor clock cycle.

Since the L1 cache miss penalty is 2 processor cycles:

\[ C_{L1} = 2M_1 \]  

(2.11)

By definition:

\[ M = m_{cyc} \cdot M_1 \]  

(2.12)

where \( m_{cyc} \) is the average amount of time (in processor cycles) that it takes for main memory to service a memory request. And:

\[ C_{L2} = s_{L2} \cdot M_1 \]  

(2.13)

where \( s_{L2} \) is the L2 cache service time (in processor cycles).

The amount of time it takes for main memory to service a processor initiated memory request is not simply the response time of main memory. As utilization of the memory bus increases (by both explicit requests from the processor and implicit loading due to cache fills), memory response time increases. We describe the overall main memory response time in terms of the ideal response time of main memory and a measure of the amount of memory bus contention:

\[ m_{cyc} = m_{cyc}^{ideal}(1 + m_{bus}) \]  

(2.14)

where \( m_{cyc}^{ideal} \) is the ideal response time (in processor cycles) of main memory and \( m_{bus} \) is an average fractional utilization of the memory bus.

Since the processor explicitly uses the memory bus only when it presents requests to main memory (implying both a L1 and L2 cache miss and subsequent fill), we approximate
the memory bus contention by the ratio of main memory requests to processor cycles.

\[ m_{bus} = \frac{(1 - h_1 - h_2)M_1}{P} \]  

(2.15)

This simplistic method for determining bus contention models the following:

- If the bus is used for a memory operation, then the entire bus is used; no memory operation requires more bandwidth than the bus supports. The bus is 32 bits wide and all memory operations operate on 32 bit words.
- The bus is clocked at the same rate as the motherboard clock and can deliver data from main memory to the processor within 1 cycle.

This model does not take into account bus usage patterns. A queuing model would be more accurate. However, since we expect that queue lengths rarely exceed one memory request, the linear approximation described should be sufficient.

Equations 2.8–2.15 constitute the general descriptive model for the test platform.

2.3 System Utilization Model

We construct a system utilization model directly from the equations developed in Section 2.2. Based on the GOOP approach (See Section 1.2, Step 3), we must find a way to construct modified general descriptive models such that the similarity in the models causes like terms to be substituted and subtracted. When the number of dependent variables is reduced enough so that the remaining system of equations is solvable either symbolically or numerically (given values for the easily measured dependent variables, i.e. time of execution), a complete system utilization model exists.

Section 2.3.1 outlines the underlying principle that allows linking general descriptive models. Section 2.3.2 shows how this principle applied to a very simple system to produces a working system utilization model. Section 2.3.3 applies the same principles to the general descriptive model of the test platform (from Section 2.2) to create a system utilization model for the test platform.

2.3.1 Underlying Principle

The system utilization model relies on the following principle:

For a given motherboard clock, all on-chip functions of a 486D-class microprocessor execute twice as fast on a 486/DX2 processor than on a 486/DX processor (since the DX2 processor is internally clock doubled). All off-chip functions execute at the same speed (using the motherboard clock) regardless of processor type.

We infer the following equations (note that \( X_{33} \) denotes that term \( X \) holds for a 486/DX-33 [processor and motherboard clock at 33 MHz]; \( X_{66} \) denotes that term \( X \) holds for a 486/DX2-66 [processor clock at 66 MHz, motherboard clock at 33 MHz]):

\[ P_{33} = 2P_{66} \]  

(2.16)

\[ C_{L1,33} = 2C_{L1,66} \]  

(2.17)
Note that Equation 2.18 holds only if all overlap instructions completed by the DX2 processor are also completed by the DX processor. We make the assumption that the normal length of an overlap sequence is short enough that this is true.

2.3.2 Simple Example

Equations 2.16–2.19 are extremely simple and intuitive, yet are the basis for creation of a system utilization model from some general descriptive model. The principle used in creating the system utilization model from the general descriptive model is best shown by example.

Assume a system (S_x) that either spends time in the processor or in main memory. Given only execution times for an instruction stream on physically modified versions of S_x, we use the system utilization model to determine the amount of time spent in each of the processor and main memory.

The general descriptive model for this system is:

$$T_x = P_x + M_x$$

(2.20)

where $T_x$ is overall time, $P_x$ is time spent in the processor, and $M_x$ is time spent in main memory.

If the instruction stream were executed on $S_x$ twice, once with a 486/DX-33 processor and once with a 486/DX2-66 processor, we break execution time down as follows:

$$T_{33} = P_{33} + M_x$$

(2.21)

$$T_{66} = P_{66} + M_x$$

(2.22)

Substituting Equation 2.16 into Equation 2.21 yields:

$$T_{33} = 2P_{66} + M_x$$

(2.23)

Since $T_{33}$ and $T_{66}$ are easily measurable (just the execution time of the instruction stream with the given processor), it is clear that the linear Equations 2.22 and 2.23 have a unique solution that gives the times spent executing in the processor ($P_{33}$ and $P_{66}$) and the time spent in main memory ($M_x$). Equations 2.22 and 2.23 represent the system utilization model for this simple system.

2.3.3 Application to Test Platform

We apply the same techniques used in Section 2.3.2 to the general descriptive model for the test platform derived in Section 2.2.

For a 486/DX2-66 processor, Equations 2.11 and 2.13 become:

$$C_{L1,66} = 2M_{1,66}$$

(2.24)

$$C_{L2} = 2M_{1,66}$$

(2.25)

since $s_{L2}$ is 2 processor cycles at 66 MHz. This implies that for a 486/DX2-66 processor,
Equations 2.8–2.10 become:

\[ T'_{66} = P_{66} + C_{L2} + M - O_{66} \]  
\[ T''_{66} = P_{66} + (1 - hL1)(C_{L2} + M - O'_{66}) \]  
\[ T^*_{66} = P_{66} + hL2 \cdot C_{L2} + (1 - hL1 - hL2)(C_{L2} + M - O'_{66}) \]  

Note that we have evaluated the \( \max(C_{L1,66}, C_{L2}) \) term in Equation 2.28 to \( C_{L2} \).

Analogous equations can be created for a 486/DX-33 processor. Using these equations and substituting terms from Equations 2.16–2.19 yields:

\[ T''_{33} = 2P_{66} + C_{L2} + M - 2O_{66} \]  
\[ T''_{33} = 2P_{66} + (1 - hL1)(C_{L2} + M - 2O'_{66}) \]  
\[ T^*_{33} = 2P_{66} + hL2 \cdot 2C_{L1,66} + (1 - hL1 - hL2)(C_{L2} + M - 2O'_{66}) \]  

Note that we have evaluated the \( \max(C_{L1,33}, C_{L2}) \) term in Equation 2.31 to \( C_{L1,33} = 2C_{L1,66} \).

Since \( m_{\text{ideal}} = 7.5 \) cycles for the 486/DX2-66 processor (The ideal memory response time (depending on synchronization) can be either 7 or 8 cycles (assuming a 66 MHz clock).

This is consistent with the total memory response time of between 107ns – 121ns with 70ns DRAM.), Equations 2.12–2.15 become:

\[ M = m_{\text{cyc}} \cdot M_{1,66} \]  
\[ m_{\text{cyc}} = 7.5(1 + m_{\text{bus}}) \]  
\[ m_{\text{bus}} = \frac{(1 - h1 - h2)M_{1,66}}{P_{66}} \]

Given \( T_{66}, T'_{66}, T''_{66}, T_{33}, T''_{33}, \) and \( T^*_{33} \) Equations 2.24–2.34 represent 11 equations with 11 unknowns. This system of equations is solvable symbolically\(^4\), and at least one solution is guaranteed to exist. Therefore, this system of equations can be used to correspond to a complete system utilization model for the test platform. The symbolic solution defines the dependent variables of the model in terms of one another. However, since none of the dependent variables are easily measured (see Section 1.1.2), this information is not directly useful.

The independent variables \( T_{66}, T'_{66}, T''_{66} \) for some fixed instruction stream can be determined by simply timing the execution time of that instruction stream on the hardware configuration corresponding to each of the 6 \( T_x \)'s. Thus, after 6 runs of the instruction stream on carefully varied hardware configurations (corresponding to the model), the system utilization model can determine, among others, the following information for the given instruction stream:

\(^4\) and I have the \( \approx 5 \) meg Maple output to prove it...
\[
P \equiv \text{total time spent actually processing in the processor}
\]
\[
h_1 \equiv \text{L1 cache hit rate}
\]
\[
h_2 \equiv \text{L2 cache hit rate}
\]
\[
h_1 \cdot C_{L1} \equiv \text{total L1 cache miss penalty time}
\]
\[
h_2 \cdot C_{L2} \equiv \text{total time spent in L2 cache}
\]
\[
(1 - h_1 - h_2)M \equiv \text{total main memory response time}
\]
\[
m_{\text{cyc}} \equiv \text{average main memory response time}
\]

The system utilization model for the test platform was developed by assuming some fixed instruction stream. We extend the idea of the instruction stream to a complete software system and obtain a hardware utilization profile for the entire system. We are now able to determine the instruction-level performance of the software system by quantifying the effects of the underlying hardware complexity, and can apply this data to guide modifications to enhance system performance.
Chapter 3

Validation with Consistency Checks

One of the steps in developing a mathematical model to describe system performance is performing a consistency check between the model outputs and some known behavior of the system. Once the model has been validated in this manner, one can use it to understand real systems where the system behavior is not completely known beforehand.

Starting from a conceptually simple point (Equation 2.5), we additional terms to the general descriptive model in stages to produce new equations such that the validity of each stage (and resulting equation) could be easily reasoned. Forming the system utilization model in Section 2.3 from the general descriptive model was a purely mathematical exercise and should represent a correct derivation if the principle assumed in Section 2.3.1 holds.

3.1 Test Properties

Three tests (model consistency checks) constructed to test the validity of the system utilization model (and hence the correctness of the underlying principle) are described in Sections 3.2-3.4. Each of the tests was written in 486 assembly code, and had the following properties:

- One could determine exact cycle counts required by the processor ($P_{pred}$) through static evaluation of the test code.
- One could determine the L1 and L2 cache hit rates ($h_{1pred}$ and $h_{2pred}$ respectively) through static evaluation of the test code and careful analysis of the behavior of the memory subsystem as described in Section 2.1.

For each test, we check model outputs ($P_{goop}$, $h_{1goop}$, and $h_{2goop}$) against their known (predicted) values for consistency. Model outputs $m_{cyc}$ and $m_{bus}$ are checked as well to ensure that they are valid values and consistent with one another.

Each test executed 6 times on the test platform as required by the system utilization model: For each of the processor types (DX/33 and DX2/66), execute the test with each of the cache states (L1 and L2 off, L1 on and L2 off, L1 and L2 both on) defined in the general descriptive model. Section 3.5 describes some limitations of the given consistency check tests.

Recall from Section 2.2.1 that $h_2$ is the hit rate of the L2 cache in terms of total memory operations. $h_2$ relates to the actual hit rate of the L2 cache ($H_2$) in terms of
memory requests presented to the L2 cache by:

\[ H2 = \frac{h2}{1 - h1} \]  

(3.1)

For tests 2 and 3, \( h2 \) is much smaller than \( H2 \), so reported errors in \( h2 \) are artificially increased relative to the error of the actual L2 cache hit rate \( (H2) \).

### 3.2 Test 1

Test 1 is a basic test for the model. Test 1 preloads a 4kB section of memory into the L1 cache. The L1 cache is 8kB, so this block fits completely within the L1 cache. In each iteration of the test, the processor sequentially reads the 4kB block (incrementing the memory address by one byte at each step); the L1 cache should handle all memory requests. See Figure 3-1 for pseudocode.

```plaintext
<preload data into L1 cache> ; load data to be used into L1 cache
<start timer> ; start the timer
for iterationCount do ; for number of iterations
  memAddr := base ; set base memory address for the data used
  until memAddr >= end ; until the entire block has been read
  read [memAddr] ; read [memAddr] (from L1 cache)
  step memAddr ; use the next memory address
  end
<stop timer> ; stop the timer
```

Figure 3-1: Pseudocode for Test 1

Table 3.1 compares raw model outputs per iteration for \( P, h1, \) and \( h2 \) with the predicted values for test 1; Table 3.2 shows system utilization per iteration (on the “normal case” DX2/66 processor with L1 and L2 caches enabled) as determined by the model outputs for test 1. We see that \( P, h1, \) and \( h2 \) are accurately determined by the model.

<table>
<thead>
<tr>
<th>Predicted</th>
<th>Model</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P ) (cycles)</td>
<td>24583</td>
<td>24404</td>
</tr>
<tr>
<td>( h1 ) (%)</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>( h2 ) (%)</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 3.1: Comparison of Predicted vs. Model outputs for Test 1

### 3.3 Test 2

Test 2 tests the model more completely than test 1 by including use of the L2 cache. Test 2 preloads a 128kB section of memory into the L2 cache. The L2 cache is 128kB so this
<table>
<thead>
<tr>
<th>Term (from Equation 2.10)</th>
<th>Time (cycles)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>24415</td>
<td>Total execution time</td>
</tr>
<tr>
<td>$P$</td>
<td>24404</td>
<td>Time spent in processor</td>
</tr>
<tr>
<td>$h_{L2} \cdot C_{L2}$</td>
<td>11</td>
<td>Time spent when L2 cache hits</td>
</tr>
<tr>
<td>$(1 - h_{L1} - h_{L2})(C_{L2} + M)$</td>
<td>0</td>
<td>Time spent when both caches miss</td>
</tr>
<tr>
<td>$(1 - h_{L1} - h_{L2})O'$</td>
<td>0</td>
<td>Time spent in overlap</td>
</tr>
</tbody>
</table>

Table 3.2: System utilization for Test 1

block fits completely within the L2 cache (note that the L1 cache contains only the last 8kB of data loaded into the L2 cache at any time). In each iteration of the test, the processor sequentially reads the 128kB block (incrementing the memory address by the size of a cache line at each step); the L2 cache should handle all data memory requests (since we are forcing the L1 cache to miss). The L1 cache will continue to service the instruction memory requests. See Figure 3-2 for pseudocode.

```plaintext
<preload data into L2 cache> ; load data to be used into L2 cache
<start timer> ; start the timer
for iterationCount do ; for number of iterations
    memAddr := base ; set base memory address for the data used
    until memAddr >= end ; until the entire block has been read
        read [memAddr] ; read [memAddr] (from L2 cache)
        step memAddr ; use the next memory address
    end
<stop timer> ; stop the timer
```

Figure 3-2: Pseudocode for Test 2

Table 3.3 compares raw model outputs per iteration for $P$, $h_1$, and $h_2$ with the predicted values for test 2; Table 3.4 shows system utilization per iteration (on the normal case DX2/66 processor with L1 and L2 caches enabled) as determined by the model outputs for test 2.

The model outputs for test 2 are less accurate than those for test 1, but the model is still able to determine the amount of time spent in the processor within 1.25%. Model outputs for $h_{1_{goop}}$ and $h_{2_{goop}}$ are slightly lower than the actual values. To account for this, note that the amount of data used in test 2 is larger than the L1 cache. As the inner loop
walks through the data for test 2, the L1 cache must eventually flush the actual instructions being executed (which are necessarily stored in the L1 cache) to the L2 cache; this causes the L2 cache to flush and reload some test data, reducing both $h_{1\text{goop}}$ and $h_{2\text{goop}}$ ($h_{1\text{pred}}$ and $h_{2\text{pred}}$ did not take this cache interaction into account).

### 3.4 Test 3

Test 3 tests the model more completely than test 2 by allowing main memory to service data requests. Test 3 preloads a 256kB section of memory into the L2 cache. The L2 cache is 128kB so this block does not fit. The L2 cache contains only the last 128kB of data loaded (and the L1 cache will contain only the last 8kB of data loaded into the L2 cache at any time.) In each iteration of the test, the processor sequentially reads the entire 256kB block (incrementing the memory address by the size of a cache line at each step), then sequentially reads the entire block in the reverse direction (decrementing the memory address by the size of a cache line at each step). The L1 cache should handle no test data requests (since we are forcing it to miss); the L2 cache should hit on exactly half the data requests (128kB worth of data whenever the direction of the data reads are reversed); main memory should service the other half. The L1 cache will continue to service the instruction memory requests. See Figure 3-3 for pseudocode.

Table 3.5 compares raw model outputs per iteration for $P$, $h_1$, and $h_2$ with the predicted values for test 2; Table 3.6 shows system utilization per iteration (on the normal case DX2/66 processor with L1 and L2 caches enabled) as determined by the model outputs for test 2.

<table>
<thead>
<tr>
<th>Term (from Equation 2.10)</th>
<th>Time (cycles)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>114808</td>
<td>Total execution time</td>
</tr>
<tr>
<td>$P$</td>
<td>49774</td>
<td>Time spent in processor</td>
</tr>
<tr>
<td>$h_{L2} \cdot C_{L2}$</td>
<td>57524</td>
<td>Time spent when L2 cache hits</td>
</tr>
<tr>
<td>$(1 - h_{L1} - h_{L2})(C_{L2} + M)$</td>
<td>9137</td>
<td>Time spent when both caches miss</td>
</tr>
<tr>
<td>$(1 - h_{L1} - h_{L2})O'$</td>
<td>1627</td>
<td>Time spent in overlap</td>
</tr>
</tbody>
</table>

Table 3.4: System utilization for Test 2

The model outputs for test 3 are less accurate than those for test 1, but the model is still able to determine the amount of time spent in the processor within .066%. We account for the minor discrepancies in model outputs for $h_{1\text{goop}}$ and $h_{2\text{goop}}$ with the same rationale used for the observed errors in test 2. Note that the percentage of memory operations serviced by

<table>
<thead>
<tr>
<th>$P$ (cycles)</th>
<th>Predicted</th>
<th>Model</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>196618</td>
<td>197933</td>
<td>0.67%</td>
</tr>
<tr>
<td>$h_1$ (%)</td>
<td>75.8</td>
<td>76.3</td>
<td>0.66%</td>
</tr>
<tr>
<td>$h_2$ (%)</td>
<td>12.1</td>
<td>11.7</td>
<td>-3.30%</td>
</tr>
<tr>
<td>$1 - h_1 - h_2$ (%)</td>
<td>12.1</td>
<td>12.0</td>
<td>-0.83%</td>
</tr>
</tbody>
</table>

Table 3.5: Comparison of Predicted vs. Model outputs for Test 3
<preload data into L2 cache> ; load data to be used into L2 cache
<start timer> ; start the timer
for iterationCount do ; for number of iterations
    memAddr := base ; set base memory address for the data used
    until memAddr >= end ; until the entire block has been read
        read [memAddr] ; read [memAddr] (from L2 cache, main mem)
        step memAddr ; use the next memory address
    end
    until memAddr < base ; reverse direction read of entire block
        read [memAddr] ; read [memAddr] (from L2 cache, main mem)
        reverse step memAddr ; use the next memory address
    end
<stop timer> ; stop the timer

Figure 3-3: Pseudocode for Test 3

<table>
<thead>
<tr>
<th>Term (from Equation 2.10)</th>
<th>Time (cycles)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>554675</td>
<td>Total execution time</td>
</tr>
<tr>
<td>$P$</td>
<td>197933</td>
<td>Time spent in processor</td>
</tr>
<tr>
<td>$h_{L2} \cdot C_{L2}$</td>
<td>68635</td>
<td>Time spent when L2 cache hits</td>
</tr>
<tr>
<td>$(1 - h_{L1} - h_{L2})(C_{L2} + M)$</td>
<td>382464</td>
<td>Time spent when both caches miss</td>
</tr>
<tr>
<td>$(1 - h_{L1} - h_{L2})O'$</td>
<td>94357</td>
<td>Time spent in overlap</td>
</tr>
</tbody>
</table>

Table 3.6: System utilization for Test 3
main memory \((1 - h1 - h2)\) has an absolute error of only 0.83%. Table 3.6 also shows that overlap execution occurs 24.7% of time spent servicing a memory request in main memory; this is consistent with analysis of the static code stream and the observed memory cycle time \((m_{cy})\) of approximately 8.8 cycles.

### 3.5 Test Limitations

As stated in Section 3.1, we created tests 1–3 partially because aspects of their hardware utilization profiles \((P, h1, \text{and } h2)\) could easily be determined statically (and thus compared with the model outputs). To facilitate this, the tests ran in a slightly contrived environment that could artificially increase the observed accuracy of the system utilization model:

- The instructions executed for all runs of any given test were *exactly* the same.
- Each test was executed in a single-processed environment where the executing thread had full control over the processor
- No I/O was performed during any test run.
- The timer only needed to be started and stopped once during any given test run.

Since these conditions may not hold in the real software systems under consideration, some consistency checks should be run against the actual system (when possible) in order to increase confidence in the validity of the model outputs.
Chapter 4

Application to Software System

This chapter illustrates how we use the system utilization model to understand and improve the performance of a compute-intensive software system by analyzing its instruction-level performance. Section 4.1 provides background information on the system under consideration. Section 4.2 describes how we apply the system utilization model to the system. Section 4.3 analyzes the instruction-level performance of the system (with respect to its system utilization). Section 4.4 shows how changes made to the system (based on the analysis in Section 4.3) modified, and improved, the instruction-level (and overall) performance profile of the system.

4.1 Background on the GVWin System

The Xerox GVWin software emulator (Guam) is a software-only virtual machine emulator that allows industry standard PCs to run binaries compiled for the Mesa processor. Guam runs over Microsoft Windows and is targeted for Intel486-based personal computers. Guam provides Mesa instruction-level emulation by implementing Mesa opcodes as microcode in the virtual machine—i.e., in 486 assembly code. Major functional divisions within Guam are [8, 3, 1]:

**Opcode Emulator (Midway)** Midway faithfully implements the Mesa Principles of Operation (PrincOps) (PrincOps is the functional specification that must be met for some virtual machine to implement the Mesa processor abstraction). This includes support for Mesa managed virtual memory; Mesa instruction fetch, decode, dispatch, and execution; Mesa process runtime support; and Mesa display bit block transfer (BitBlt). Midway is written in 32-bit x86 assembly code [11, 9].

**Windows Agents** Early hardware implementations of the Mesa processor used specialized hardware I/O processors (IOPs) to handle I/O activity. Guam is implemented strictly in software and relies on Windows to provide all hardware level I/O handling. The term agent refers to a module written in 32-bit C code that serves as an interface between the Windows API and the Mesa world [3].

**Pilot Operating System** Pilot provides all application-visible runtime support to application programs written for the Mesa processor (in this case, the Guam/Midway implementation). Only public interfaces within Pilot can be used by application programs; modules existing privately within the Pilot kernel can be modified without
regard for effect on applications. All Windows agents in the PC world are mirrored by Pilot heads in the Mesa world. Pilot is written in the Mesa programming language [8].

A frequency distribution of Mesa instructions weighted by Intel486 cycle counts per emulated opcode predicted Midway would require an average of 29 Intel486 cycles to execute one Mesa instruction\(^1\) (this does not include I/O time) [9]. After extensive optimization (using a multitude of software optimization and analysis techniques), between 45 and 60 cycles per Mesa instruction were observed. Measuring and analyzing Midway subsystem execution times reveal that Guam is not spending a disproportionate amount of time performing any particular task.\(^2\) Further analysis and subsequent modification to the Guam system takes into account the following considerations:

- Midway code is tight within the scope of its current design. instruction-level optimization (based on 486 cycle counts) will not yield any further performance increase.

- Binary compatibility with all existing Mesa applications is essential. Any new version of Guam must faithfully implement Mesa PrincOps and modifications to Pilot must maintain the correctness of all public interfaces. For these reasons, modifications of Pilot will not be considered (making analysis of Pilot performance unnecessary for this project).

- Agents are merely interfaces between Windows and the Mesa world. Optimization of any agent within the scope of its current design will not yield any substantial performance increase.

4.2 Applying the System Utilization Model

As stated in Section 4.1, Midway performance is more than a factor of 1.5 slower than the predicted levels. We apply the system utilization model created in Section 2.3 to the Guam system in an effort to understand the instruction-level behavior of Midway.

The system utilization model requires that execution time for some fixed instruction stream be measured. We use two separate Mesa-based test scenarios as test runs for analysis with the model: Pilot booting (BOOT) and document pagination (PAG). BOOT and PAG were chosen because they are not inherently display intensive procedures (display is handled by both Midway and Windows). Thus, the model analyzes the performance of the computation/emulation engine as opposed to the display subsystem. Since we are concerned only with the performance of the Midway emulator (and not the entire Guam system), special timing code was added to the Guam system (see Figure 4-1): Initialize a timer when one of the test runs starts; whenever execution enters the Midway subsystem, measure elapsed time; when the test is complete, save the total elapsed time data. Note that this implementation correctly causes I/O (especially disk accesses) to not be timed since all I/O is handled by Windows.

\(^1\)Similar techniques have been used to correctly predict the performance of hardware-based Mesa PrincOps emulators.

\(^2\)It has been said that “there is not something wrong [with Guam]... the whole thing is just not right.” [3]
These timers are sufficient, since Windows provides a non-preemptive multitasking environment. Once Midway has control of the processor, no other thread can execute instructions without Midway explicitly giving up control.\(^3\) Note that implementing software timers within the system being analyzed can affect the hardware utilization (by adding memory references, processor cycles, etc.). However, for the Guam system, Midway execution is interrupted on the order of 300 times per second; for a DX2/66 processor, Midway will execute on average for approximately 220000 cycles before losing control of the processor. The effect of the multiple timer calls is thus negligible.

### 4.3 Performance of Baseline GVWin

Utilizing the timing code described in Section 4.2, we collect data regarding execution time of Midway for the BOOT and PAG tests in accordance with the system utilization model. Tables 4.1 and 4.2 show how time was spent during each test run (both in overall numbers of seconds and per emulated Mesa instruction).

<table>
<thead>
<tr>
<th>Time Spent In...</th>
<th>Overall (s)</th>
<th>Per Instruction (cycles)</th>
<th>% of Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>283.3</td>
<td>51.4</td>
<td>--</td>
</tr>
<tr>
<td>Processor</td>
<td>170.5</td>
<td>30.9</td>
<td>60.1</td>
</tr>
<tr>
<td>L2 Cache Hits</td>
<td>17.1</td>
<td>3.1</td>
<td>6.0</td>
</tr>
<tr>
<td>Main Memory</td>
<td>88.5</td>
<td>16.1</td>
<td>31.2</td>
</tr>
</tbody>
</table>

Table 4.1: Performance profile for baseline Midway (BOOT)

Analysis of the hardware utilization profile for both tests suggests that the original prediction of 29 cycles per emulated opcode was in fact close to the actual time spent processing; the original predictions were off by a factor of at least 1.5 because they did not

---

\(^3\) Except in the case of system faults that DOS/Windows must handle. We make the assumption that these events are rare compared to the total execution time of Midway over any given test run.
<table>
<thead>
<tr>
<th>Time Spent In...</th>
<th>Overall (s)</th>
<th>Per Instruction (cycles)</th>
<th>% of Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>61.1</td>
<td>61.0</td>
<td>-</td>
</tr>
<tr>
<td>Processor</td>
<td>31.8</td>
<td>31.7</td>
<td>52.0</td>
</tr>
<tr>
<td>L2 Cache Hits</td>
<td>5.7</td>
<td>5.7</td>
<td>9.3</td>
</tr>
<tr>
<td>Main Memory</td>
<td>21.4</td>
<td>21.4</td>
<td>35.0</td>
</tr>
</tbody>
</table>

Table 4.2: Performance profile for baseline Midway (PAG)

take into account the time spent handling L1 and L2 cache misses. Table 4.3 lists the cache hit rates for each test.

<table>
<thead>
<tr>
<th></th>
<th>BOOT</th>
<th>PAG</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_1$ (%)</td>
<td>91.2</td>
<td>89.1</td>
</tr>
<tr>
<td>$H_2$ (%)</td>
<td>48.9</td>
<td>56.9</td>
</tr>
<tr>
<td>$(1 - h_1 - h_2)$ (%)</td>
<td>4.5</td>
<td>4.7</td>
</tr>
</tbody>
</table>

Table 4.3: Cache hit rates for baseline Midway

4.4 Performance of Modified GVWin

Understanding Midway's instruction-level performance profile based on its hardware utilization provides the information necessary to guide modifications that improve emulator performance. Based on the results indicated in Section 4.3, it is clear that improving the cache performance of the Midway emulator can provide a substantial performance improvement.

Midway execution causes memory references in the following basic ways:

- Memory references caused directly by individual Mesa opcodes.
- Storage for internal data structures.
- Actual Midway code being executed on the processor.

Memory references caused directly by individual Mesa opcodes are not controllable by Midway. Since these references are based on the binaries executing in the emulated Mesa world, we can do nothing to improve cache performance of this set of memory references. Data structures (and associated algorithms) as well as actual code being executed can be modified to exhibit better spatial and temporal locality in order to improve cache performance. In general, we accomplish this by shrinking the size of the working set of memory required over some period of time; however, in order to do so, extra time spent processing is usually required.

We modify the Guam system so that Midway would exhibit better cache performance.\footnote{Specific implementation details are not discussed. Rewriting the Midway emulator was accomplished...}
• Compress (or remove entirely) most large or often accessed data structures. Reducing the size of the working set of the data requires additional processing (based on 486 cycle counts) in most cases.

• Whenever possible, within reason, share code segments instead of repeating them. The increased use of jump instructions (which are relatively expensive in terms of processor cycles), fixup code, and procedure calls (instead of inlined code) requires additional processing (based on 486 cycle counts) in most cases.

Tables 4.4 and 4.5 show how time was spent during each test run (both in overall numbers of seconds and per emulated Mesa instruction) in the modified Midway emulator.

<table>
<thead>
<tr>
<th>Time Spent In...</th>
<th>Overall (s)</th>
<th>Per Instruction (cycles)</th>
<th>% of Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>259.4</td>
<td>47.1</td>
<td>–</td>
</tr>
<tr>
<td>Processor</td>
<td>187.5</td>
<td>34.0</td>
<td>72.3</td>
</tr>
<tr>
<td>L2 Cache Hits</td>
<td>14.8</td>
<td>2.7</td>
<td>5.5</td>
</tr>
<tr>
<td>Main Memory</td>
<td>57.2</td>
<td>10.4</td>
<td>22.1</td>
</tr>
</tbody>
</table>

Table 4.4: Performance profile for modified Midway (BOOT)

<table>
<thead>
<tr>
<th>Time Spent In...</th>
<th>Overall (s)</th>
<th>Per Instruction (cycles)</th>
<th>% of Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>56.5</td>
<td>56.6</td>
<td>–</td>
</tr>
<tr>
<td>Processor</td>
<td>36.6</td>
<td>36.7</td>
<td>64.7</td>
</tr>
<tr>
<td>L2 Cache Hits</td>
<td>4.7</td>
<td>4.7</td>
<td>8.3</td>
</tr>
<tr>
<td>Main Memory</td>
<td>13.8</td>
<td>13.8</td>
<td>24.4</td>
</tr>
</tbody>
</table>

Table 4.5: Performance profile for modified Midway (PAG)

Table 4.6 lists the cache hit rates for each test running on the modified Midway emulator. Note that the modified version of the Midway emulator spends considerably more time in

<table>
<thead>
<tr>
<th>Cache Hit Rate</th>
<th>BOOT</th>
<th>PAG</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h1$ (%)</td>
<td>94.2</td>
<td>92.6</td>
</tr>
<tr>
<td>$H2$ (%)</td>
<td>56.9</td>
<td>62.2</td>
</tr>
<tr>
<td>$(1 - h1 - h2)$ (%)</td>
<td>2.5</td>
<td>3.8</td>
</tr>
</tbody>
</table>

Table 4.6: Cache hit rates for modified Midway

the processor than the baseline (due to the “cache performance versus processor cycles through application of standard programming techniques (to improve cache performance) coupled with creative programming ideas and a lot of hard work. The entire coding process was not quite as easy as it sounds, and is not relevant to the discussion regarding the system utilization model.
"tradeoff" that was made); its overall execution speed is faster, however, due to much better cache performance (and thus less time spent in main memory).

Table 4.7 and 4.8 compare the performance profiles of the baseline and modified Midway emulators with respect to the number of cycles needed to execute a Mesa instruction.

<table>
<thead>
<tr>
<th>Time Spent In...</th>
<th>Baseline</th>
<th>Modified</th>
<th>% Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>51.4</td>
<td>47.1</td>
<td>-8.4%</td>
</tr>
<tr>
<td>Processor</td>
<td>30.9</td>
<td>34.0</td>
<td>+10.0%</td>
</tr>
<tr>
<td>L2 Cache Hits</td>
<td>3.1</td>
<td>2.7</td>
<td>-12.9%</td>
</tr>
<tr>
<td>Main Memory</td>
<td>16.1</td>
<td>10.4</td>
<td>-35.4%</td>
</tr>
</tbody>
</table>

Table 4.7: Compare Baseline v. Modified Midway Performance (BOOT)

<table>
<thead>
<tr>
<th>Time Spent In...</th>
<th>Baseline</th>
<th>Modified</th>
<th>% Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>61.0</td>
<td>56.6</td>
<td>-7.2%</td>
</tr>
<tr>
<td>Processor</td>
<td>31.7</td>
<td>36.7</td>
<td>+9.5%</td>
</tr>
<tr>
<td>L2 Cache Hits</td>
<td>5.7</td>
<td>4.7</td>
<td>-17.5%</td>
</tr>
<tr>
<td>Main Memory</td>
<td>21.4</td>
<td>13.8</td>
<td>-35.5%</td>
</tr>
</tbody>
</table>

Table 4.8: Compare Baseline v. Modified Midway Performance (PAG)
Chapter 5

Conclusions

5.1 Summary

The GOOP approach to modeling system utilization provides a software-only method to determine instruction-level performance characteristics of software by measuring only timing data on different execution runs of the software system. The data available by applying the model is a useful guide in modifying the software system to improve overall performance; these modifications are based on information that was not previously easily attainable.

The GOOP approach extends the idea of simply creating descriptive mathematical models of systems to linking multiple descriptive models. After building a general descriptive model, modify the hardware slightly and create a new model describing the modified hardware. Because of the similarity in models, terms will substitute and subtract thereby reducing the number of dependent variables. By repeating this process, eventually the system of models will be solvable (at least numerically), and the dependent variables can be determined by applying easily measured values for the independent variables, i.e. time of execution.

We tested this approach by building a general descriptive model for an Intel-486 based PC with an IBM 433D motherboard. Relying on the fact that on-chip activities execute twice as fast on a 486DX/2 processor than on a 486DX processor (with the same motherboard clock), we are able to link two general descriptive models for the same PC (with the different processors). The result is a system of 11 equations with 11 unknowns that has a guaranteed solution. We applied this system of equations, the system utilization model, to infer the low-level performance of an existing software system. Based on the hardware utilization profile obtained (main memory was a performance bottleneck), we modified an actual software system to exhibit increased performance (by causing the caches to hit more, at the expense of increased processor usage) in a way that was not obvious prior to understanding the instruction-level performance of the system.

5.2 Evaluation

The GOOP approach to system utilization modeling developed in this thesis provides an inexpensive, portable method for determining low-level utilization of hardware sub-elements by a software system. Once one constructs a system utilization model for a given platform, it can be applied to virtually any software system executing on that platform.

Although it is unclear exactly how accurate the resulting model is when applied to real
systems, we are convinced that the model outputs are within an acceptable error; the goal is not to determine absolutely how time is being spent during system execution—rather, it is to gain an understanding of how time is being spent. This understanding guides performance enhancing modifications.

The system utilization model is most useful for analysis of compute-intensive software systems. The model cannot be directly applied to those systems for which I/O time (disk access, network, display, etc.) cannot be factored out of the measured execution time. If one could find a way to add these I/O terms to the general descriptive model and link multiple descriptive models so that these terms also substitute and subtract, the GOOP approach would be able to model the I/O as well. The GOOP approach may not apply to some systems that have a high dependency on fine-grained absolute time. For example, if some software system needed to perform a task every \( n \) processor cycles, that task would occur more times per instruction with the caches disabled or with the 486DX processor than with the caches enabled or with the 486DX/2 processor. If the task overhead is high (compared with the number of instructions executed), the different models would not necessarily be running the same, or even similar, instruction streams. Obviously for the same reasons, if the instruction stream exhibits a large degree of non-determinism, then the GOOP approach may not be applicable.

5.3 Future Work

Determining and improving the accuracy of the model itself is neither impossible nor unnecessary. Constructing validation tests that exercise the model more fully under a larger range of varying conditions would allow a more confident assessment of the accuracy of the model. If the model could be fine tuned enough, it may be useful for actually analyzing the effects of different hardware configurations to determine the best hardware platform for some software system (as opposed to simply guiding software modifications). It is possible that the approach may eventually be applied towards guiding hardware modifications.

Currently, the GOOP approach to modeling system utilization is an analytic descriptive approach. It requires that the system already exists. If easy and accurate ways of measuring and predicting some of the dependent variables are explored and created, then the system utilization model constructed could be extended to be predictive as well. It could then serve as a useful tool in preliminary software design.
Bibliography


