Quasistatic Computing Environments

by

Ian S. Eslick

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Engineering

and

Bachelor of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1996

Author .....
Department of Electrical Engineering and Computer Science
May 24, 1996

Certified by .....
Thomas F. Knight
Senior Research Scientist
Thesis Supervisor

Accepted by .....
F. R. Morgenthaler
Chairman, Departmental Committee on Graduate Thesis

Quasistatic Computing Environments

by

Ian S. Eslick

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 1996, in partial fulfillment of the
requirements for the degrees of
Master of Engineering
and
Bachelor of Science in Electrical Engineering and Computer Science

Abstract

Modern compiler optimizations can be roughly characterized as *Static* or *Dynamic.* Static optimizations are often limited by a lack of knowledge of run-time program behavior. Dynamic optimizations seek to circumvent this problem by delaying instruction bindings until run-time values are known. Much of the analysis for dynamic optimizations must be performed statically as any run-time analysis is in the critical path of the program and reduces the benefit of the optimized code. A model of *Quasistatic* computation addresses this difficulty by including compilation as an integral part of a program's lifetime. Run-time feedback from previous program executions are used during compile-time analysis to produce speculative optimizations based on expected run-time values. Additionally, expensive off-line analysis can be performed without impacting run-time performance. Quasistatic applications may continually adapt to changing data sets and environments. This thesis introduce a systems framework for enabling quasistatic computation and explores implications of quasistatic computing environments.

Thesis Supervisor: Thomas F. Knight
Title: Senior Research Scientist

Acknowledgments

It seems strange, that a single short page is all we have to express to the world the magnificent contributions others have made to the course of our work and our lives. The acknowledgements I wish to relate would completely dwarf the material in this thesis.

First and foremost I owe a incredible debt of gratitude to my parents for their years of patient support and unconditional acceptance. Without the intellectual and emotional freedom their care provided, I could never have come so far.

A heartfelt thank you goes to Shannon Marie Prescott for her friendship and her many examples of living passionately. Thanks in kind to Ed Frodel for helping to point the way to an incredible journey of self-discovery. Additional thanks to all the wonderful teachers I've had over the years who have nurtured and encouraged me.

To the crowds of friends and classmates from 1W, 5E, ESG, ESP, Leadershape, Coffeehouse and to all the other wonderful, supportive people from within the Institute; Anne Hunter, Karen Sollins, Travis Merrit and others. A special thanks to Stormy Charles for weathering my many outbursts and darker moods and always being there for me. To Peter Dourmashkin for helping me to learn how to think.

And how can I forget all those in the lab who have been helpful in diverting and enlightening me? To the whole crowd from the 7th floor and my comrades from "The Halting Problem" - I'm going to miss you guys!

And finally, a huge thank you to those who have had the largest impact on this work. My thesis advisor Tom Knight has been amazingly supportive and encouraging and made my time here both interesting and motivating. I would like to extend a great measure of appreciation and respect to Andre for his role as friend, fellow student and mentor over the years. I cannot count the ways that my interactions with Andre have effected my life, my thinking and the directions I've chosen to take. May your incredible creativity, dedication and caring spirit bring you all that you wish out of life.

Contents

List of Figures

Chapter 1

Introduction

This thesis integrates a new computation model, Computational Quasistatics[1], into a system framework, a Quasistatic Computing Environment, leading to a tightly-integrated model of a computer operating system. This system enables increased efficiency for applications, system routines and the management of system resources.

This thesis introduces the notion of a Quasistatic Computing Environment and explores its advantages and tradeoffs. Computational Quasistatics allows the computer system to achieve higher application performance by performing more aggressive optimization and adapting the application executable to regularities in the program's run-time characteristics. The enabling addition to our existing models of computation is the explicit inclusion of automated run-time feedback information and recompilation into the deployed lifetime of an application. This inclusion implies the existence of certain capabilities in the compiler, the execution environment and the methods of program dissemination.

A Quasistatic Computing Environment, then, embodies the system support and design perspective required to utilize and take advantage of the insights developed in Computational Quasistatics. It allows the compiler to use data about run-time characteristics in off-line analysis to automatically produce higher performing executables.

[1]Computational Quasistatics was first introduced in [7]

1.1 Compilers and Compilation

In abstract, a compiler seeks to translate from one language to another, reorganizing the destination language for the most space and/or time efficient implementation of the source language. We do this because we wish to describe our computations using a more abstract, or compact formulation than the primitive instructions provided by target languages of interest: generally microprocessor instruction sets.

To accomplish this, it is necessary to understand the underlying issues which govern the methods and efficiency of the compilation process:

- A model of computation
- Representations of computation
- What the compiler knows
- A model of the target language

1.1.1 Models of Computation

In general terms, we can categorize computing into two major methods: static and dynamic. Static computation is characterized by predictable control and data flow, access patterns and timing. Static computation can be easily exploited by compilers to minimize execution latency by restructuring this easily analyzable form of computation. Dynamic computation, by contrast, is characterized by unpredictability: frequent changes in control flow, random data access patterns and unknown dependencies make analysis difficult. Most often, dynamic computation arises from dependencies on program inputs which change as the user requirements and datasets change. An interpreter, for example, is highly dynamic.

Static Computation

Static computations are those which can be resolved at compile time, independent of run-time, or dynamic, data. Examples include static type information, static

instruction sequences (no data-dependent branching) and static memory allocation (pre-allocated global data structures, etc).

In most procedural languages the binding of functions or subroutines is completely static, such that all the bindings can be resolved at compile-time, potentially enabling further optimization. Additionally the primitive operators in procedural languages (again, Fortran and C-like languages) are completely determined at compile time.

Dynamic Computation

Dynamic computation is characterized by run-time binding and run-time, data-dependent decisions. Examples include dynamic typing, as in LISP-like languages, dynamically allocated memory (such as C's malloc), dynamic linking, exception handling and address translation (ie virtual memory systems).

We also find features of our programming languages which exhibit dynamic properties; dynamic method resolution (C++, LISP), pointers into dynamic structures which must be traversed at run time, aliasing[2] and the dynamic latency of most superscalar processors with their multiple, interacting execution units, deep pipelines and speculative execution.

Quasistatic Computation

In section 3 we will explore the quasistatic model in detail. In essence the quasistatic computing model can be thought of as a third type of computation - a process by which data-dependent (traditionally dynamic computation) and control intensive computation can be converted between application executions into a more efficient coding optimized around expected run-time characteristics.

One way to view this process is as a compression problem. By using run-time information available to the executing program, gathered over multiple runs in an off-line decision and analysis phase, we can reorganize the code to run the fastest for the observed common cases, much like Huffman coding in compression theory.

[2]pointers in general can only have a fully determined dereferenced value at run-time and multiple pointers can share the data pointed to allow for unpredictable interactions

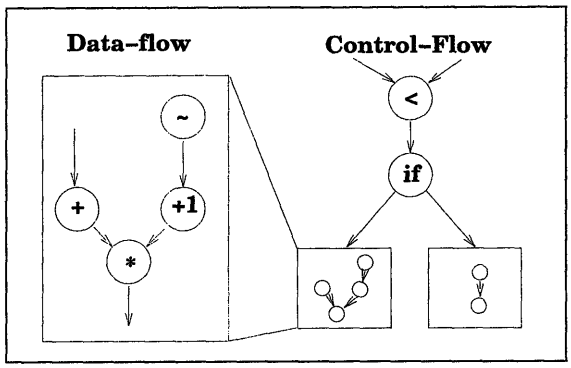

Figure 1-1: Simple fragment of a control/data flow graph

Supporting this form of computation, however, requires much system context. Addressing this problem is the primary subject of this thesis.

1.1.2 Representations of computation

To deal with these three methodologies we can use the standard compiler control/data flow representation. In essence we convert the logical program structure of the source language, via a set of simple transforms[3] to a set of primitive instruction "nodes" such as adds, and multiplies for several data types, dynamic control nodes such as comparisons which lead to a branching between nodes. This set of nodes is interconnected via directional graph edges which establish the data dependencies in the program. This graph is usually referred to as a data/control flow graph or an intermediate representation. Figure 1-1 shows a pictorial representation of a small piece of a dataflow graph.

To handle the restrictions of dynamic computations, sections of purely static dataflow are chunked together into basic blocks. Each basic block is bounded on one end or the other by a dynamic computation, generally a conditional branch or entry/exit from a function.

This is the technique commonly used in compilers which handle static and dy-

[3]This ignores, for the sake of minimizing complexity, advanced source language analysis such type inference, formal verification, etc.

namic computing styles. Compilers, because they don't know anything about the run-time behavior can make nothing but worst case assumptions about the dynamic computation portion of the intermediate representation.

Quasistatic computation allows the compiler to utilize run-time information and convert much of what is commonly unoptimizable, dynamic computation into quasistatic computation.

To do this the representation requires at least one added feature: the ability to add annotations. These annotations maintain information about run-time characteristics as well as to embed state information from a higher-level compilation process which is driving the quasistatic computation.

1.1.3 What the compiler knows

This leads us to ask what the compiler knows. Usually the intermediate graphs have only primitive compute nodes, and so the compilation throws away much of the higher-level structure associated with the source language. While eventually this is necessary, it often happens after the front-end translation and keeps the optimization engine from using this data.

Secondly, the compiler only knows what is embedded in the intermediate form. This includes only the known and unknown dependency structure of the primitive program description. Unfortunately this is only half the picture. Dynamic computing decision are made by incoming, run-time data. The assumption, implicit in the quasistatic computing model, is that some of this data input remain the same between program runs. This information needs to be included in the information available to the compiler if higher performance is to be extracted by the optimization engine.

1.1.4 Modeling the target language

Finally, there is the need to model the target language to facilitate translation. Just as we would like to leverage information from the source language, when possible, we would also like to be able to optimize to specific characteristics of the target language.

One widely-used compiler, gcc[17], developed by the Gnu project uses a backend description language which contains information about instruction types, instruction costs, available registers and beneficial instruction orderings. Modern processors, the most common target languages, are rife with complex dependencies between instructions and delays for certain instructions.

In abstract (the actual process is much nastier), a translation pass turns the intermediate graph into a pseudo representation of the target language, commonly known as RTL (Register Transaction Language).

From this format the compiler uses its built in optimization passes to efficiently allocate machine registers, order instructions, fill branch-delay slots, perform primitive branch prediction (one way of reducing the cost of dynamic computation) and efficiently convert the RTL instructions to target language instructions.

A research compiler system SUIF[18] (Stanford University Intermediate Format), used in the substrate of the experimental QCE, takes a different model. Intended for use in research systems, SUIF defines a combined set of intermediate formats called high-SUIF and low-SUIF which are a combination of a high-level intermediate graph and an RTL. One interesting feature of SUIF is that it maintains many of the higher level control constructs such as loops and case statements for explicit analysis by the optimization systems.

Here the backends are written by hand, and most commonly the SUIF format is converted back to C for compilation by gcc. The important issue here is that efficient backend compilation passes are difficult to build and generally operate on a very primitive representation, such as RTL or low-SUIF.

1.2 Trends in Modern Computing Systems

Today's computing environment is in constant flux. Hardware platforms are likely to change every 6 to 18 months, applications are constantly being upgraded and environmental conditions such as network throughput and latency vary with the time of day. The underlying technology moves even faster: compiler enhancements can improve

compiled code, program feature sets change quickly and algorithm development is in many cases outpacing the Moore's law hardware speedup. The only constant in the industry is continual, rapid change.

Yet, the traditional, still dominant, paradigm of software development and deployment ignore these facts. There is an implicit assumption that deployed applications need be rarely updated and that the execution environment is predominantly static.

These assumptions can be seen in a quick analysis of the traditional development framework, which usually includes the C programming language and an OS such as UNIX or Windows. A program is developed by a software house and at some point is deemed "releasable" after undergoing beta tests. At this point a binary executable is released to the user population. This released version is never free of bugs, yet any upgrades happen on the order of months or more for large-scale commercial products. In the user's environment the operating system may be upgraded, the machine architecture will change, peripherals will be added and in multi-tasking machines, the available resources will vary with time. Even with all this variance, a fixed, single application is deemed sufficient for all users.

Neglecting to adapt to end-system conditions requires programmers and compilers to make conservative, general-case assumptions - hiding performance behind artificial abstractions.

Many of the roots of these problem are historical, maintained by industry momentum and the fixed executable software model of modern operating systems. Time-to-market pressures restrict optimization efforts and today's compilation technology is primarily concerned with static program analysis, when optimization is bothered with at all - the danger of program bugs introduced by optimizations in the compiler motivate most compiler designers to be conservative.

Commercial products are concerned with a number of program characteristics. The most obvious are feature sets, development time and execution efficiency. The need to focus on execution efficiency has grown less relevant in recent years as the need for clear, maintainable code and short time-to-market has dominated, forcing programmers to spend less time optimizing their programs.

Evidence of this can be found in every system in wide use today: 50 megabyte word processors which require 100 Megabyte operating systems and 200Mhz, 16 Megabyte home computers to run. In general, it is the end user who suffers, either through pitiful application performance or by being forced to purchase a faster hardware platform.

When program optimization is performed, it is usually accompanied by a simple profile phase which points out the critical sections of the code which are then hand-optimized by programmers.

Extracting some key trends we see:

- Increasing system complexity and market pressures reduce the time and effort available to optimize programs
- In practice, program optimizations are limited to those based on static analysis
- Deployed programs are static, retaining their structure despite changes in environment costs and user data sets
- Program binaries are one-size-fits-all, seemingly optimized for performance across all possible datasets
- Any use of profiling data is by programmers and gathered on sample datasets or standard benchmarks

These trends point out a prevalent problem in software and system development: how to enable and produce complex, well-structured systems that execute efficiently in the presence of a changing environment without paying penalties in programmer development time.

Chapter 2

Background

2.1 Traditional Optimization

Traditional optimization techniques focus on static computation. A full study of the formalisms behind these traditional techniques can be found in [2]. To provide some background on these techniques, this section will go into further detail on a few key optimizations and mechanisms used to perform the analysis of a program's intermediate representation.

Common optimizations include:

- Dead code elimination - Prune provably unreachable nodes from the intermediate graphs
- Constant propagation - Take any variable assigned constant values and inline them directly, skipping the variable reference and performing partial inlining when possible
- Loop hoisting - Take code provably independent on iteration variables and side-effect free (i.e. no volatile pointer accesses or other side effecting function calls) and move it outside the loop construct
- Operator reduction - Reduce complicated operations to simpler operations via range analysis and simple power-of-two reductions (division and multiplication

by power-of-2 constants reduces to static shifts)

- Register variables - Assign critical variables to be stored in registers instead of memory

- Loop unrolling - Rather than taking the branch each cycle, the compiler lays out multiple segments of the loop, optimizes between them and handles the loop variable increment explicitly

To perform dead code elimination a standard technique is to use an algorithm known as a relaxation algorithm which propagates sets of connected nodes up and down the graph. If a node is disconnected, say by a branching node where the arguments to the branch test are constant, then the node can be pruned from the tree.

For constant propagation we can perform a simple flow analysis, where we propagate the constant assignment source down all dependent nodes where the variable is deemed "live". As references to the variable are found they can be converted to constant references. Combined with operator reduction and dead-code elimination, this can be a very beneficial technique.

A second class of optimizations are usually performed during backend translation. Often these are called peephole optimizations as they often involve very specific, very simple motion of target instructions to fill branch delay slots and the like.

The most important datum to remember about static optimization techniques is that they are completely inhibited by the presence of any dynamic dependencies. The presence of pointers is even worse, since we can assume nothing about what they point to and whether what they point to changes during a function call.

Traditional medium-level programming languages like C are rife with these limiting features. Dynamic language such as Scheme and Common Lisp face a similar problem with the explicit features of dynamic typing, garbage collection and dynamic variable/function binding.

2.2 Run-time Optimization

Computational performance is often limited by the dynamic computation patterns in programs of which the compiler has no knowledge. Because dynamic computations are often dependent on input to the program, knowing run-time values increases the potential scope and benefit of optimization. Some systems gather this data at run-time and generate code on-the-fly which is specifically optimized around the run-time values encountered. Examples include Synthesis [16], Dynamic Code Generation[8] and Deferred Compilation[14]. This form of optimization is limited by the need to simplify run-time analysis and code generation so it does not adversely effect performance.

In the Synthesis operating system kernel, Massalin and Pu perform several program performance optimizations that allow them to utilize knowledge about run-time behavior to reduce the overall run-time of a program. A good example of their techniques lie in user IO handling. Rather than having a single piece of code dedicated to reading and writing all files, the Synthesis kernel generates custom code for each file that is opened.

In CMU's deferred compilation, compile-time analysis leads to the creation of specialized code generators that produce object code when run-time data is known. Dawson's dynamic code generation works in a similar way. The compiler identifies when run-time optimizations are likely to be beneficial and arranges to produce output code when run-time values are known.

All of these methods have shown modest improvements, but the cost of run-time code generation often overshadows the benefit of the resulting code.

2.3 Profile-Based Optimization

In a 1971 paper, Donald Knuth proposed that "The 'ideal system of the future' will keep profiles associated with source programs, using the frequency counts in virtually all phases of a program's life."[13] Knuth found in his studies of Fortran programs that

most programs spent more than 1/2 of the time during a program execution in only 4% of the program code. His group discovered that a dramatic 4-5x speedup was possible by optimizing only this critical portion of the code. However the techniques used here were primarily by hand, though the use of a profile-aware optimizing compiler was mentioned.

More recently, specific methods have been proposed and implemented which use information gathered at run-time to make compile-time decisions about how to structure compiled code. Some methods include branch prediction [11], trace-scheduling [10] and specific performance optimizations such as CustoMalloc [12]. None of these systems yet automate the entire process. Rather they rely on the user to specify when data is to be collected and when it is to be used by the compiler.

2.4 Execution Profiling

The acquisition of lightweight feedback in a selective manner is a research issue that has been addressed by a variety of academic and commercial researchers. Feedback may be categorized into one of the following:

- Statistical profiling
- Basic block counting
- Call graph annotation
- Variable binning
- Timing
- Environment information

Ideally each of these feedback methods would be enabled at different times during the programs lifecycle to help focus feedback analysis, enable beneficial optimizations and reduce profiling costs. For example, feedback directed specialization of program functions use call graph annotation and variable binning to determine which functions

have a nearly static parameter call pattern and can produce functions optimized around those parameters. The specialized functions can conditionally be called when general parameters match the those used in the specialized functions.

In all cases, we must understand when, where, and how much feedback is needed for various optimizations and analysis and understand the relevant costs. Before this question can be properly answered it is important to understand what optimizations are going to consume feedback and how data about the optimizations need to be represented.

2.5 Statistical Profiling

Statistical PC sampling, can be accomplished with very little ($< 10\%$) performance impact[9]. On a sun sparcstation, informal experiments have shown that this overhead can be as cheap as 3%[3].

Statistical sampling is performed by an external process, usually interrupt driven, which samples the application's program counter at periodic intervals. Over a long-enough run, we have a statistical trace which shows where the program spends most of its time.

Usually the program spends a majority of its time ($> 80\%$) in one or two key functions. Once a statistical profile has been acquired, more expensive profiling methods can be directed at those portions of the program which are most relevant to the program's overall efficiency.

2.5.1 Basic Block Profiling

Basic block profiling is a much more detailed, explicit method of acquiring a knowledge of where the program spends its time. A counter is added to each basic block which counts the number of times that basic block is executed. This data, while incredibly expensive to implement (adding a memory reference to each basic block, often adding a factor of 2 to the profile being performed) can be used for a variety of interesting optimizations, and is often used in branch prediction.

2.5.2 Call Graph Annotation

Call graph annotation is similar to basic block counting, but adds less overall overhead. Each function call site is affixed with a counter which determines the number of times a particular function was called from a particular site.

2.5.3 Variable binning

A less common profiling technique is variable value analysis. Here a set of memory locations are allocated to some variable at some point in the program. Each time the program passes through this point, the variable value is tested against previously used bins, if it is equal to the value in a bin, the bin's counter is incremented, otherwise it is placed in an empty bin. When the bins are full, an "overflow" bin is used to count the number of exceptions.

This information can be used to determine rough estimates of a variable's dynamic probability density function; that is how much of the time does the variable under consideration take on a particular value. If the overflow bin dominates, the variable contains high entropy, or randomness. However, if we have a limited set of variable that occur with high regularity, we can exploit this knowledge through function or block specialization (section 3.2).

2.5.4 Cycle Counting

The most accurate of timing methods, which requires special hardware features to implement, is cycle counting. The method is quite simple in theory, a hardware counter starts running (or the continually running value is read) upon entry to a piece of code (preface to a function call, beginning of a loop, etc) and upon exit the value of the counter is read. This tells the system exactly how many cycles it took to execute that function. In general this is an incredibly good metric of time. If the variable value isn't swapped out during a context switch, context switching can completely skew the results.

This is the method of choice for making accurate comparisons of the advantages

provided by different optimizations as we can get an exact count of cycles. This is especially important in modern processors as instruction execution latency is highly variable and very hard to model in the compiler backend.

2.5.5 Environment Information

Environment information consists of information about machine load, machine type, user information, configuration, network load, scheduling priorities and other machine statistics that might effect program running behavior. This information is used by optimization passes which seek to develop correlations with optimization benefit and surrounding environment conditions.

One example application is an analysis of the impact of cache size. If a series of optimizations is increasing the size of an inner loop, and the size of that loop becomes large enough to flow out of the processor's cache, then we will see a performance degradation instead of an enhancement. This information needs to be made available to the compilation system so that such dependencies can be handled.

2.6 Modern Operating Systems

What is an operating system? Through conversations with other researchers, it seems commonly agreed upon that the answer to this question is less clear than it once was. Often, when we think of operating systems images of Microsoft DOS, Windows, UNIX and the Macintosh come to mind. Informally, operating systems are understood to be the set of applications and utilities that enable a user to interact with a physical hardware platform. This particular view glosses over a tremendous body of functionality that takes place inside what is commonly considered an operating system - the low-level models of protection, hardware interaction, program execution and peripheral driver support.

For the purpose of this thesis, the term operating system will refer to the basic services required for applications and service-layers to utilize and interface to the user's hardware. This includes support for file, network and user IO as well as multitasking

the underlying processor or compute engine.

However, applications no longer live in a purely operating system dominated world. Rather users have access to a wide array of machines and information services.

In the mid 1990's the computer world underwent a drastic shift in the focus of both academic research and commercial and development.

In the academic world money and energy shifted away from the high-performance, parallel-machine world with its model of tightly-coupled processing elements in vast, expensive arrays to embrace distributed computing and clusters of workstation as the latest hot topic. Much of this can be seen as ARPA shifted its funding emphasis to take interest in commodity technologies. Ubiquitous, cheap and commodity become the research technology focus.

In the commercial world we have seen Microsoft, the largest vendor of operating systems in the world announce that it had become "a network company". Sun unveiled its Java model. Other companies introduced notions of a "network computer", internet box and the much-hyped set-top boxes.

In short it is clear that networking technologies, the Internet and the World Wide Web (WWW) will play an incredibly strong role in the structure of tomorrow's operating systems. This thesis uses the term *operating environment* to refer to the collection of services, machines and users which is becoming the mainstream computing model.

2.7 Search Techniques

We can formulate the set of all possible program translations as a configuration space. This immediately brings to mind the notion of search as as a fundamental method for finding program translations with particular characteristics (minimum expected execution times, minimum executable size, etc).

However as we see in the work by Massalin [15], brute force searching of sequences of processor instructions for even small dataflow graphs generates immense search spaces. Clearly we can borrow from some of the past few decades of work from AI in the formulation and searching of complex spaces. The work described in chapter 6 is

based on some of the ideas in heuristic search.

Traditional heuristic search uses an evaluation function to determine how close a step in a search is to the goal state. The search process follows the sequence of branches, taking the next state that minimizes this function. The heuristic function allows the search procedure to take advantage of domain specific information.

The problem with pure, algorithmic solutions such as this is that they do not translate well to problems that do not easily decompose for specification as a search space. Both the space representation and the search algorithm need to be specialized for such problems.

Chapter 3

Computational Quasistatics

Quasistatic computation has features of both static and dynamic computing. The computational framework enables adaptation of the application to end-system and end-user characteristics through application profiling on end-user datasets and automatic recompilation. When a program is specialized to the datasets it encounters in practice, less computing needs to be accomplished for the program to accomplish its tasks, reducing the execution time of the program.

While computational quasistatics builds off existing work in profile-based optimization, it goes a step further than traditional techniques and introduces a feedback path between the run-time execution of an application and the compiler as an explicit component of the computational model. This feedback path is used to pass information about an application's actual deployed behavior to the compiler for use in subsequent compiler runs. This feedback can be used to enhance static decisions and make optimistic (yet safe) assumptions about actual data sets. Over time, the data is collected in aggregates to provide feedback on which values are likely to be constant from run to run. This feedback path allows the compiler to take advantage of run-time information previously unavailable.

The recompilation portion of the quasistatic cycle may happen off-line or in parallel with program execution, but is assumed not to interfere with the program run-time. This offline compilation may use all the static analysis available in modern compilers but can also use run-time information to make smarter predictions, specialize por-

tions of code to expected values and select potential alternative implementations of code segments. Dynamic optimizations may also be used in cases where there is no run-to-run correspondence between values.

This section discusses the key issues involved in quasistatic computation. The following section expands on this background and widens the scope to incorporate all the system-level mechanisms required to support quasistatics.

3.1 Fundamentals

As mentioned in the introduction, quasistatic computing uses feedback information to speculatively convert commonly occuring dynamic computations to static computations. As discussed below (section 3.2), specialization is the enabling primitive operation. Speculatively applying specialization may be looked at in several different ways.

Elimination of Unused Functionality By assuming certain features in the input data stream or the surrounding environment, the compiler is able to remove excess functionality associated with the fully general support coded into the program's description. This is, in essence, the techniques described for dynamic, on-line code generation - however the optimization takes place off-line, increasing the potential benefit.

Efficient common-case execution For known data values or ranges, it is possible to re-optimize the code to perform better on the common-cases.

To achieve these advantages it is necessary to collect feedback data over *multiple* runs. Optimizing common values from previous runs can have a negative effect if the patterns optimized around change from run to run. The term quasistatics implies that such off-line optimization is slowly varying over long epochs. We can then perform cheap optimizations on values that are constant between runs. For more quickly varying data patterns, the compiler can apply run-time optimizations when and where they are deemed to provide a performance advantage.

3.2 Specialization

We look at code specialization as the key enabler that takes our dynamic computation and transforms it so that we can apply purely static analysis. Treated in greater depth by Brown in [3], specialization takes run-time values which are known to have very low or even zero entropy at run-time and treats them as constants. Once this is done, the optimizing compiler places a dynamic check which ensures that the variables hold the accessed value and then perform our standard static analysis, such as constant propagation and dead-code elimination, to produce a compressed program. If the value fails the comparison, the original code is run.

Specialization can be applied to a wide set of programming primitives. Basic blocks can be specialized, functions may be specialized; in fact any time an analysis is performed to determine what computation to apply to some input data, the analysis can be reduced or ignored and the resulting computation performed directly if the data is already known.

3.3 Specific techniques

Specialization generalizes to domains beyond simple control-flow constructs. Techniques such as Custo-malloc, which utilize knowledge about system routines, or other higher level program or system characteristics may also be performed by a quasistatic compiler.

In Custo-malloc, knowledge about standard uses and costs of the malloc function in C programs is leveraged to enable automated optimization of memory allocation. In general, anytime a kernel call has a cumulative effect, such as allocating more and more chunks of a system resource, knowing how much resources are allocated over the lifetime of the program allows the costs of repeated kernel calls to be reduced by performing one call and distributing allocated resources in user space. This is advantageous because of the incredible performance impact of operating system context switching.

The costs and tradeoffs in our optimization strategies will continue to change and shift with technology. If the compiler is not able to accommodate this higher-level knowledge, then it will become outdated in little time.

3.4 System-level support

Performing quasistatic optimizations requires certain support from the surrounding system. The core of the compiler requires profiles to make decisions on what to optimize and how to optimize. The optimization functions need a common interface to manipulate the applications they are optimizing.

In addition to this core compiler/optimizer support, it is essential to take into account the requirements of the surrounding system. Creating multiple copies of programs to specialize programs to either user and/or machine characteristics has impacts on both the program loading mechanism and the storage requirements of the server. Multiple program instances require more management capabilities and the decision about when such specialization is important must be made.

As we see, there are a variety of system-level concerns in the application of computational quasistatics to real world situations; this is the subject of the next chapter.

Chapter 4

A Quasistatic Computing Environment

This chapter discusses QCE's at an abstract level, nailing down terminology and the roles the system will fulfill. It provides us with the motivation and framework for the following chapters which explore a concrete design.

4.1 Characterization

From our background discussions we know that our computer systems consist of a wide variety of hardware, software functionality and coupling to other systems. For the sake of discussion in this thesis, operating systems consist of the software which provide an application interface to the hardware on which the machine runs. In today's systems where distributed storage and service servers are common, the operating system is no-longer a single-machine affair. In dealing with this distributed environment we add a new layer to our discussion and call the set of services which manages interaction with remote machines an operating "environment".

To be complete, we notice that in any production 'operating system', there is a set of tools that are presented to the user which provide a set of basic services on top of the operating system layer. For the sake of further discussion we refer to such a collection of tools as applications and the user interface. These services are not

crucial the model presented below and will only receive cursory discussion.

Our QCE is intended to be deployed in an environment which contains multiple, heterogeneous computers and multiple users. While single-machine, single-user models are possible, such systems are becoming less common. With the advent of the previously mentioned "network boxes" it is likely that the application model will include distribution from central locations via the network to user-interface boxes.

4.2 Role and Responsibilities

What does a QCE do? We start from the system-level requirements we can extract from our model of Computational Quasistatics:

- Compilation support
  - Feedback techniques
  - Manipulable intermediate representation
  - Support for new optimizations
- Low-level support
  - Store feedback data
  - Store intermediate program representations
- Higher-level support
  - Integration of feedback data into program representation
  - Scheduling of analysis
  - Scheduling of recompilation
  - Resource utilization constraints

Because there is software in the environment layer which is dealing with the management of shared system resources and optimizing the intermediate program representation under the Quasistatics model, we can no longer separate the compiler and the operating system.

4.2.1 Compilation Support

The job of the traditional compiler in this framework is greatly reduced, while the logical role of the compiler has been expanded. To explain, instead of providing optimized object code, the compiler (acting as a front end translator) produces an intermediate program description[1], removing the backend translation and optimization from the compiler. From this point on, optimization of the code is the responsibility of the QCE.

For development purposes, it makes sense to retain the traditional model as it provides for faster turnaround and easier debugging. The compilation model discussed here is intended for release-ready code.

4.2.2 Execution

When the intermediate format is introduced to the system for the first time it is registered with a "site manager" which stores the description in the database, makes the new program visible to the user and compiles an executable for platforms supported at this site. This initial compilation includes statistical profiling which is the first step in the quasistatic compilation process.

When a user runs the program, profile information is kept throughout the duration of the program run. Upon completion, this information is sent to the site manager[2] where it is processed and the resulting information is back-annotated into the program intermediate representation.

In addition to program feedback, the system tracks the user, runtime, machine, machine characteristics, system characteristics and any large-scale system information (such as network or processor load) which may be relevant to the analysis.

[1]This has implications for software distribution and protection which are addressed in section 7

[2]this sending of data implies some mechanism for assembling and transmitting the data, but the specific mechanism used is implementation specific. The method I used is described in section 5.

4.2.3 Analysis and Recompilation

The next phase in the program's life is a compiler analysis. An "Analysis Engine" forms the core technology which enables our QCE - everything else is data management and distribution.

First, the site manager, in addition to managing applications, contains a set of analysis/optimization modules. These modules are intended to be stand-alone in that new modules may be added to the system and be immediately integrated into the compilation process. Each module consists of:

- An analysis pass - performs a minimal computation to determine the potential benefit of the program optimization and whether more feedback is required for benefit or accuracy of analysis and the expect cost of that feedback
- An optimization pass - using feedback data, the optimization pass reorganizes the program representation
- Feedback placement - based on what was determined in the analysis phase, the module may initiate further feedback acquisition.

The job of the analysis engine is to determine which of these modules to use in the analysis phase and, based on the result of the analysis, to determine the ordered set of optimization passes to apply during recompilation.

However trusting the programmer's intuition about what data is really needed and what the actual speedup is likely to be is usually a bad choice. It is a well-known fact that most programmers have little knowledge about the actual cost tradeoffs in their code - the traditional observation is that in register allocation of function variables the compiler often made the better choice than the programmer.

To handle this problem, we add an additional layer of information management to the system, requiring it to keep track of the accuracy of the analysis phase and the actual performance of the optimization passes. This can be difficult as we have conflicting requirements. We want to quickly search the space of possible optimizations

to find a highly efficient translation, yet at the same time it is hard to separate out the effects and interactions of various optimizations.

As this problem is both central to the design of a QCE and fraught with complexity, we discuss this portion of the system in depth in section 6.

Finally, once the set of optimizations has been determined the program is schedule for recompilation.

4.2.4 System Management

Because of the potential for code explosion, version explosion and the computational impact of the analysis and recompilation, it is essential for the QCE to be conscious of resource limitations and to strive for minimal impact.

Managed resources include:

- Computing "cycles"
- Database storage space
- Network load

One of the "no-intrusion" features of QCE is the utilization of idle cycles at low-priority for analysis and recompilation. This serves two purposes, it keeps the user from noticing the impact of the QCE on their individual workstation performance and allows us to use otherwise "wasted" cycles on tasks which will increase the overall system efficiency. This implies the ability to distribute any of these analysis or compilation modules to user machines in the network.

If the QCE has a limited amount of storage space for it's intermediate code, optimization module and task working space - then it needs to make tradeoffs between increased specialization, code size and application diversity (per-machine, per-user versions, etc).

An additional resource, the network, is also important. During times of intensive use, overuse of the network for disseminating compilation tasks or integrating feedback

can adversely impact network utilization by users. In these cases data may be cached on the remote execution platforms and analysis can be differed to "off-hours".

Chapter 5

Implementation Architecture

This chapter discusses the prototype system framework used to explore QCE's. Pseudo-code is used where appropriate to describe interfaces and/or algorithms. The system is designed and partially implemented in C++, so the pseudo-code will mimic the semantics of C++ objects.

A diagram of the envisioned implementation can be found in figure 5-1. Here we see the system broken down into a series of discrete code components:

- Database storage
- Application support code
- Machine server
- Site server

The database storage handles all the storage requirements for the site server, allowing access into program representation, code optimization modules and working data associated with the QCE. The application support code is responsible for dumping the generated feedback data with enough information to allow it to be re-integrated with the stored intermediate representation. It also informs the machine server of application start and completion times.

The machine server is responsible for handling the acquisition and optional caching of all feedback generated by the application. It provides information to the site server

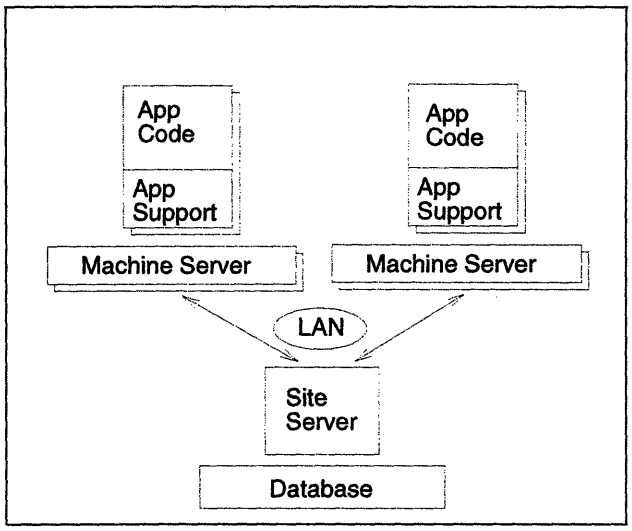

Figure 5-1: QSE: Structural View

about system and application characteristics. The machine server detects idle cycles and allows the site server to download scripts which can be run on the machine server's host to accomplish background analysis and/or compilation.

As we see in figure 5-2, the site server can be further broken down into a series of logical interfaces:

- Registration
- Analysis and Learning
- Compilation & Feedback
- System Resource Information and Constraints

Several distinct subsystems implement these interfaces.

- System Information Management
- Scheduler
- Analysis.

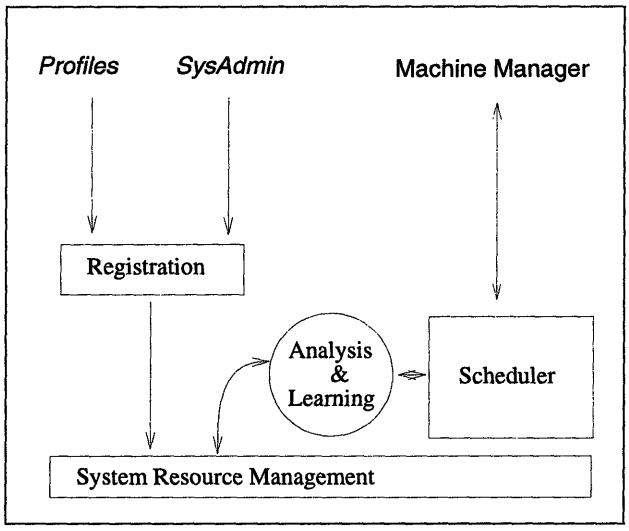

Figure 5-2: QSE: Breakdown of Site Server

The system information management portion of the site server keeps track of all data associated with programs, workstations, the server and the analysis modules. The scheduler determines what will be run and when, while the analysis phase performs the key analyses required to make the QCE fulfill its function in increasing application performance. All other portions of the system, aside from the scheduler, are directly geared to support the analysis phase. The analysis phase will be covered in detail in chapter 6.

5.1 Database

The database is necessary to maintain and provide an index into all information required by the server and its constituent processes. Ideally the database would be directly integrated into the server framework with a solid object-oriented database to reduce the effort to build in non-transparent interfaces. After working with a number of freely available systems, it is clear that to save effort and enhance reliability a simple interface to the file system is best. In a production system a database with standard features of logging, transaction management and a decent interface is necessary.

Rather than build specific interfaces through a low-level, read/write protocol with

```
DB : CoreObject
  method DBInStream GetInputStream
  method DBOutStream GetOutputStream

DBStream : Object
  method DBStream::StatusEnum Status
  method bool CloseStream

DBInStream : DBStream
  method DBInStream OpenStream(name)
  method int Read

DBOutStream : DBStream
  method DBOutStream OpenStream(name, mode)
  method bool Write(int)
```

Figure 5-3: Prototype Database Interface

all the problem inherent in file naming/management, etc. The experimental system implements a pseudo-persistent object interface. Each first-class but non-core object in the server framework has two methods which converts the object's internal state to a binary stream and vice versa. We discuss the implementation of this methodology in section 5.3.1.

Thus the primary interface to the prototype's database is via named data streams, which are handled by a stream and IO class which, in turn, are part of the core system support code.

5.2 Machine Server

The machine server plays several roles in the prototype QCE. Its primary responsibilities include:

Register programs Whenever programs which have been compiled by a QCE-aware compiler are run, they register themselves with the machine server so that program usage statistics may be maintained. In addition, when a program is compiled

for the first time, the machine server tells the site server that a new program is being added to the management list.

Gather and submit profiling data Programs which have profiling enabled in some capability or another will produce large annotated dumps containing the profiling data. The machine server takes this data, processes it and send the data along a protocol stream to the site server. It also includes in each upload information on the executing environment, the platform, total program running time and user information.

Detect idle cycles, machine characteristics The machine server watches the architecture on which it runs and identifies when the machine load is low and cycles are available for background analysis or compile support. In future versions it will also look for changes in the platform, such as new running daemons, architectural upgrades such as memory or addition of a new CPU head for symmetric multiprocessing workstations.

Serve as compute server When idle cycles are available, the machine server informs the site server of its load status. The site server can then download scripts to be run on the machine server's host platform. These will always be run at low priority.

The current profiling system is based on work-in-progress inside the SUIF compiler system by Jeremy Brown of MIT's Reinventing Computing group and, as such, the details of the profile data dump are still being refined. However, scripts which process the data and re-annotate the source SUIF are provided in this experimental system and can easily be integrated into the machine server's framework.

5.3 Site Server Components

The site server lies at the center of the Quasistatic Computing Environment. Its role is to centralize all information for its host network and provide users with a

transparent, yet high-performance interface to its supported application.

5.3.1 Implementation Features

As mentioned earlier, the whole system is architected with ease-of-use in mind to reduce the complexity of maintainability and extensibility. Each non-core object in the system is based off a base class object named, surprisingly enough, "Object". The core objects consist of:

- Database interface - Upon server startup, the database interface (fig. 5-3) is instantiated and pointed to the file system records which contain all the index information for the pseudo-persistent program being run.

- Thread manager - The thread manager is started up with an initial thread supplied as an argument. Each thread is a simple class with a *thread_main* method. The thread manager provides an abstraction to MIT's pthreads package.

- Object manager - The object manager keeps a short record of every first-class object in the system. First-class objects are all those objects necessary to contain the entire state of the system. Other objects which are temporary or subsumed by first-class objects (when the subsuming object is saved, all the dependent objects are as well - such as a single-class linked list abstraction).

The entire server state is periodically checkpointed and saved to disk for fault tolerance purposes. To accomplish this, the thread manager waits for all currently active threads to quiesce and calls a save method on the object manager. The object manager then creates a output stream into the database and dumps the state of each object in the system.

Pointers are resolved by a lookup into the object managers object table. However, to make this scheme work, a strict object model is required unless an object managing anonymous memory or standard memory storage primitives handles itself, the translation into and out of the working format.

When the log stream is used to restore the server state, the object manager creates an object that uses the stream to re-create its state at log time.

5.3.2 System Information Management

The program and system management portion of the server maintains information on system configuration information and global constraints as well as program specific information, links to relevant information and user statistics.

The system information is retained to allow the analysis and scheduling portions of the server make proper decisions in whether to spawn remote computation, add an additional specialized application, etc. The system information retained includes:

- List of Registered Applications
  - List of Users and Usage Counts
  - List of Specialized Apps - Ptr to Intermediate Format
- List of Resources
  - List of Registered Machines (running machine managers)
  - Daily Idle Compute Time
  - Available Disk Space
- Administrator/User Specified Constraints
  - Priority Applications
  - Space limits

5.3.3 Scheduling

The scheduling portion of the server maintains two important priority queues:

- Analyses to run
- Compilations to run

The analyses are run periodically for each registered application depending on how often the applications are used by users, any priorities assigned by the system administrator and how recently an analysis was run. After a job is run off the top of the queue, all jobs in a queue are resorted according to these priorities. Note that a job is not scheduled until the application has been run at least three times to get a decent sample of profiles. Without multiple runs, the accuracy of extracting quasistatic values is very low.

Ranking is currently determined in a straightforward manner. Here p is the priority level, 5 is the normal priority on a scale from 1 to 10 while u is a constant of users per month also normalized across all applications to a 1 to 10 scale.

t is the time in eight hour increments from the previous run. This assures that even the lowest priority application is run at least once a week. t is reset after the application is run. The ordering function then is:

$x < y$ iff $(t_x + p_x + u_x) < (t_y + p_y + u_y)$

One problem with such a scheme arises if it takes longer to run an analysis or compilation on all applications than the allotted week. If this happens than the priority scheme ceases to have an effect and it becomes a strict ordering based on the application run least recently.

While any ordering function is possible, other methods haven't been explored due to the lack of a complete prototype. Scheduling will prove an interesting avenue for further research in the context of a full-system.

5.3.4 Analysis

The analysis phase for each program is activated by the scheduling queue. Chapter 6 discusses what happens for each application in detail.

Chapter 6

Analysis and Optimization

The core of the QCE is the collection of analysis procedures, optimizations and learning techniques that allows individual applications to be optimized throughout their lifetimes.

Reviewing the function of these various phases, we find that to initiate a step of the optimization search process there needs to be an analysis phase which determines what optimizations are going to provide the maximum advantage for the next step of the optimization. The optimization phase actually transforms the program while the learning phase allows the system to evaluate its effectiveness and to learn to perform the optimizations faster and more effectively in the future.

Learning in this environment must take place on two fronts; with optimizations and with applications. While the system is searching for the current optimal optimization set for a particular application, it is also collecting information about the viability and advantages of the optimizations it applies to all managed programs.

Most of the operational mechanics of the QCE can be explained through understanding how knowledge is represented and used. We explore these mechanics through the underlying representations as well as the enabling processes.

6.1 Process

The overall process of compilation proceeds in three steps; analysis, optimization and evaluation. The analysis phase uses a series of estimators, defined by both the optimization pass designer and produced by the system itself; based on past experience. The first task of the analysis phase is to determine the effectiveness of the previous search step. If the performance increase was sufficient, the next step is taken from this node. Otherwise the search process backtracks to the previous state and attempts to take the second highest ranking step. The ranking from each forward step is maintained from optimization pass to optimization pass.

Once it is time to determine the next step, the estimators are evaluated and weighted together to select the next best step in the search. Once the next step has been determined, the optimization and profile acquisition lists are updated and sent to the scheduler to be scheduled. As feedback from the new executable comes into the system, it is written back into the intermediate format and statistical profile information and raw timings are fed into the program information storage unit for use in the next analysis pass.

For a complicated search process, the search methodology is almost entirely dependent on choosing the appropriate representation to represent a point in the search space and on the evaluation functions which select the next test state.

6.2 Estimators

Each estimator takes as input both the server information about the application as well as a pointer into the intermediate program representation. The return by each estimator is an expected percentage speedup. Each application has associated with it two estimators as in figure 6-1.

- Supplied
- Application-specific
- System

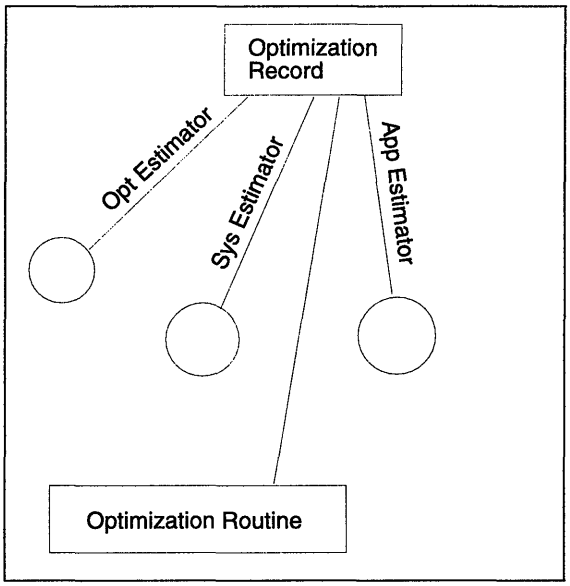

Figure 6-1: Data Structure Pictorial for Optimizations

The supplied estimators were built by the programmer to supply the programmer's expectation of total speedup. However we don't wish to trust the programmer alone. Because an application may benefit from a particular optimization in one function, but not another, the next step never ignores a particular optimization, but it does keep track of what functions and blocks it has been used to optimize, under what conditions and what the results were.

The estimators are used, much as in the scheduling metric in section 5.3.3, by summing their weighted averages. Initially, the overall benefit is averaged among all three estimates. As the system learns more and more about the supplied vs. the histogram-based estimators, it begins to skew the answer to include the influence of the more accurate method. Simply, as the percentage confidence in an estimator increases relative to another, their total contribution is skewed in a like manner.

Usually, the estimators often require actual feedback and/or system information to produce an estimated speedup, as in figure 6-2. This implies a kind of delay slot between the feedback and the optimization. A current step has to be done based on the information that can be gleaned, the next step is then performed based on the new feedback information. Again, this may require backtracking or continue from the

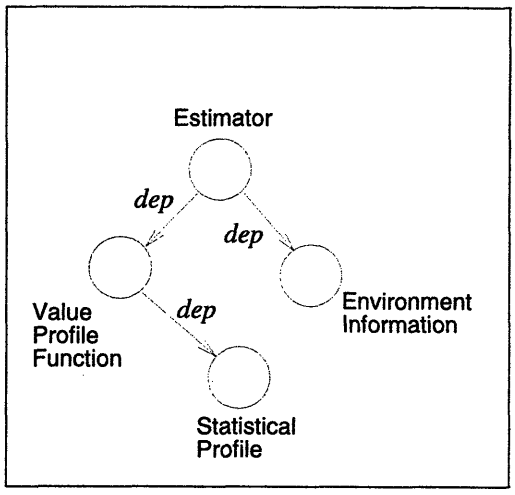

Figure 6-2: Estimators and Optimizations have Dependencies

previous step. Whether to optimize or to wait while additional profiling is done is one policy decision selectable by the site administrator.

6.3 Optimizations

If we have a continuous state space, such as geographic location, we can use a simple representation such as a latitude/longitude coordinate system as our state representation. In the fragmented, inhomogeneous configuration space of possible optimization sequences, we use primarily a functional representation; just the sequence of optimization steps to perform. As an adjunct to this optimization list, we include a list of nodes in the program intermediate storage format which are to be profiled on the next compilation pass.

In the prototype system the actual optimizations are stand-alone programs which operate over SUIF files, which are intermediate representations of the programs being optimized. Each pass converts the SUIF to a new, optimized form or adds a profiling step and outputs a new SUIF file available for further optimization.

There are several ways to search this space:

- Add one optimization at a time

- Add multiple optimizations at a time

Adding one optimization at a time makes it easy to determine the effect of each optimization, however it also makes it difficult to search the overall optimization space in an interesting time period. Even a high priority application won't be optimized more than one or two times a day, and that's assuming that the program is run that often.

For this reason, adding multiple optimizations each stage is desirable. One method that can be used for optimizations which do not optimize exactly the same code, is to time each section and determine it's addition to the total time savings. Beyond this, statistical methods are required.

A feature which allows us to compress the time required to search the entire space without resorting to complex statistical methods, is the ability to combine optimizations. Anytime the system discovers two optimizations that interact in a synergistic manner it can create a new optimization module which merely calls the two component optimizations in order.

In this way we can expand the set of useful optimizations that are actually applied and walk the space much more quickly. Methods of discovering

6.4 Histograms and Learning

After the optimization there are several brands of statistics gathered to enable the application and system evaluators to be updated.

- Estimated speedup vs. measured timing
- Overall speedup for optimization step

From these measurements we can answer some crucial questions to our learning and optimization process:

- How accurate was this estimator?

- How beneficial is this optimization in total speedup?
- What was the overall speedup and was it a linear combination of what we expected or a compound of sequencing two or more optimizations.

The answer to the first question determines the weights used to determine the effect on total scoring of each estimator for each optimization. The benefit in total actual speedup is used to build the per-application estimator.

Answering the third question requires determining whether the contribution of an optimization is a linear increase, providing the normal expected independent benefit or is experiencing a multiplicative effect with another optimization. Anytime an anomaly is discovered, it is noted as part of the application specific information stored by the site server. ie:

```
<Opt1> increased after <Opt2> by an extra X%
```

or

```
<Opt1> decreased after <Opt2> by Y%
```

After each analysis phase, the system searches these tables for the application. After a series of optimization passes and observations like the above, the system takes all three occurances and creates a new node including both Opt1 and Opt2, which is independently evaluated. The static evaluators are still used, but are multiplied by the average of the expected percentage increase expected via the interaction of the two optimization strategies.

In this manner, large spaces can, over time, be searched quickly as the most efficient collection of optimizations are applied up front.

Chapter 7

Implications for Future Computing

Now that we understand what a Quasistatic Computing Environment is, how it works, the capabilities it provides and the tradeoffs it requires; we can take a speculative look into the future to see how the QCE fits into the evolving computing landscape.

7.1 Ubiquitous Network Connectivity

The most hyped change in the computer world today is the introduction of and the long term potential for a global-scale, ubiquitous network. Constant high-bandwidth connectivity from any place on the globe is being planned and anticipated by an large number of people and commercial organizations.

With this infrastructure in place, global scale extensions to the QCE model are possible. In this model, site servers are able to communicate their discovered information about compilation methods, applications and machines to other site servers. One method for doing this is outlined in the *Global Cooperative Computing* (GlobalCC) model [6][4][1][5] proposed by MIT's Reinventing Computing group.

GlobalCC adds the notion of a "canonical server" which maintains a central repository of information about code modules (and optimization/analysis modules). Globally-distributed systems can access the server's database to download learned in-

formation about module performance in applications, the reliability of certain code, automatically generated bug reports from program crashes, profile information, etc.

In this manner a wide variety of site servers can automatically share data improving both the efficiency of the learning cycle. In addition, information on bugs and performance bottlenecks can be transmitted back to the developers. The developer can fix the bug, or change the algorithm which implements the bottleneck code and upload a new version to the server.

7.2 Changing Computing Landscape

As a result of this expectation of pervasive network connectivity, the model of what constitutes a workstation, an application and an operating system is undergoing some drastic changes. The academic world has for some time been looking distributed systems operating on clusters of workstations, but the introduction and popularization of the World Wide Web has placed a powerful commercial impetus for the world to take advantage of this network.

Current local area network system technologies aim work on loosely coupled operating system components that can be implemented on small, multi-node platforms, and distributed in an RPC manner across multiple workstation or PC servers.

Further excitement is being generated in the arena of network boxes which server primarily as high-powered, graphic terminals. The Web may very well encourage the commoditization of a machine much like the X-terminals envisioned in the 80's.

This infrastructure enables a new model of application distribution and execution where portions of the application are calculated and cached on the remote machine while the bulk of storage and calculation takes place on central server banks. If computing nodes for such servers are commoditized in either a parallel computing model or a distributed, but stripped-down model in much the way stacks of modem have been packaged into dense boxes with commodity multi-modem boards.

A QCE operating on the server side can customize applications to aggregates of actual datasets or even to specific users who pay for additional performance. The

balance of computation and the efficiency of communication can be enhanced by adding higher level transforms to the compiler which have knowledge of the existence of this network link and the importance of optimizing for minimum network transit time.

Such systems may allow both a new model of application development where local and remote load balancing is worked into the base of the application yet where the application is developed in an abstract manner and optimized by the underlying QCE instead of wasting programmer time optimizing to expected conditions.

7.3 Reliability and Time-to-market

Going back to the GlobalCC model, we can make some further projections enabled by the coupling of the QCE and canonical server models implicit in GlobalCC.

One stipulation of the GlobalCC system is that canonical server clients periodically poll the canonical server. When the client is a site server it can receive notification of the bug fix, automatically download the new version and recompile the application; transparently providing an incredibly quick upgrade turnaround. As a time-to-market and customer satisfaction enabling technology, this framework has tremendous potential.

From the perspective of a developer, the ability to quickly tie together applications and distribute them to beta users, getting almost instantaneous feedback. In addition the server can be used by the development team from any place connected to the network, allowing comments, revisions and testing feedback to be integrated together with custom interfaces (implying additional development clients, potentially including Java applets) or transparently over the Web.

7.4 Reconfigurable Computing

Another fast-growing area of research is in the arena of custom computing machines, also known as reconfigurable computing. In reconfigurable computing a computa-

tional accelerator such as a Field Programmable Gate Array (FPGA - an array of bit-level, programmable logic cells originally intended for logic emulation) is coupled with the microprocessor and programmed at run-time for an application specific hardware accelerator. While the overhead often keeps them from reaching the performance of ASIC devices, it has been shown in a variety of settings that these fine-grained and other, coarser-grained research devices can outperform general supercomputers on highly regular tasks.

With GlobalCC we enable accelerator code to be compiled in for architectures which have such devices, and for machines without such devices, the original software solutions can be compiled in.

If hardware is added to the system, it can be detected by the site manager and within hours to days, the applications that can benefit from the new hardware will automatically adapt themselves (within the system constraints, of course) to the new hardware.

Fast turnaround, transparency and high potential performance increases may very well compel future system developers to begin adopting this technology.

Chapter 8

Conclusion

This thesis has shown how integrating the Quasistatic Computing model into a Quasistatic Computing Environment leads to a tightly-coupled model of a computer operating system which enables increased efficiency for applications and the utilization of system resources.

The ability of a quasistatic computing system to specialize programs to dataset and environment characteristics allows for greatly improved application performance. The allocation of spare cycles and storage area to program optimization in combination with more efficient executable programs more efficiently manages system resources.

The complexities involved in designing and implementing such a system have been addressed in some detail. We have gained an understanding of the requirements of such systems and the support of those requirements in a real-world context.

We understand the tradeoffs involved in breaking the compiler, OS barrier and allowing the system to enable optimizations which cross any abstraction barrier in the system. When more widely adopted, such a system opens up new research potentials in feedback-enabled optimizations. With solid research QCEs and growing commercial viability, we are likely to see a growth in higher-level and higher-cost optimizations that can be used in specific situation in which they can be determined to be of use.

Finally, in the last chapter we explored the implications of such systems for future

computing; how the movement away from a one-user, one-executable model heralded by the expansion of the internet, the escalating costs of both program production and run-time and the incorporation of a global scale information framework indicate that elements of a QCE may both make practical sense and become a logical necessity to reduce cost, enhance performance and improve product reliability in the future.

Bibliography

[1] Michael Afergan, Jake Harris, Nathan Williams, Ian Eslick, and André DeHon. Global cooperative computing: Pragmatics revisited. Transit Note 126, MIT Artificial Intelligence Laboratory, March 1995.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Co., 1986.

[3] Jeremy Brown. Feedback directed specialization of c. Master's thesis, MIT, 545 Technology Sq., Cambridge, MA 02139, June 1995.

[4] André DeHon, Jeremy Brown, Ian Eslick, Jake Harris, Lara Karbiner, and Thomas F. Knight Jr. Global cooperative computing. Transit Note 109, MIT Artificial Intelligence Laboratory, October 1994.

[5] André DeHon, Jeremy Brown, Ian Eslick, Jake Harris, Lara Karbiner, and Jr. Thomas F. Knight. Global cooperative computing. In *The Second International World-Wide Web Conference 1994*, October 1994.

[6] André DeHon, Jeremy Brown, Ian Eslick, and Thomas F. Knight, Jr. Global cooperative computing. Transit Note 105, MIT Artificial Intelligence Laboratory, April 1994.

[7] André DeHon and Ian Eslick. Computational quasistatics. Transit Note 103, MIT Artificial Intelligence Laboratory, March 1994.

[8] Dawson R. Engler and Todd A. Proebsting. Dcg: An efficient retargetable dynamic code generation system. contact: todd@cs.arizonda.edu, engler@lcs.mit.edu, November 1993.

[9] Jay Fenlason and Richard Stallman. *GNU gprof.* Free Software Foundation, Inc., 675 Massachusetts Avenue, Cambridge, MA 02139, January 1993.

[10] Joseph Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[11] Joseph A. Fisher and Stefan M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *ASPLOS V*, pages 85–95. ACM, ACM, October 1992.

[12] Dick Grunwald and Benjamin Zorn. Customalloc: Efficient synthesized memory allocators. CU-CS 602-92, University of Colorado, Department of Computer Science, University of Colorado, Boulder, Colorado, July 1992.

[13] Donal E. Knuth. Empirical study of fortran programs. *Software Practice and Experience*, 1(1):105–133, 1971.

[14] Mark Leone and Peter Lee. Deferred compilation: The automation of run-time code generation. CMU-CS 93-225, Carnegie-Mellon, December 1993.

[15] Henry Massalin. Superoptimizer – a look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–126. ACM, IEEE Computer Society¡ Press, October 1987.

[16] Henry Massalin and Calton Pu. Threads and input/output in the synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 191–201. ACM, December 1989.

[17] Richard Stallman. *Using and Porting GNU CC.* Free Software Foundation, Inc., 675 Massachusetts Avenue, Cambridge, MA 02139, October 1993.

[18] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary Hall, Monica Lam, and John Hennessy. An overview of the suif compiler system. Available via Anonymous FTP as suif.stanford.edu:pub/suif/suif-overview.ps.