SRC Technical Note

1997-028

November, 1997

1997 SRC Summer Intern Projects

Compiled by James Mason

Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301
http://www.research.digital.com/SRC/

This document features informal reports by interns who spent the summer of 1997 working with researchers at DIGITAL Systems Research Center (SRC). The interns were graduate students in computer science or electrial engineering Ph.D. programs. Each worked for about three months at SRC, collaborating on a project with the research staff. The primary goal of this technical note is to describe the summer research projects. However, the interns were encouraged to write their reports in whatever format or style they preferred, so that non-technical observations (such as background and impressions arising from their stay) could also be included.

7. A High speed Interface from PCI to FireWire
   Oskar Mencer
8. SPHINX: Site-specific Processors for Html INformation eXtraction
   Robert Miller
9. Vis: A Unified Graphical User Interface For DCPI
   Robert O'Callahan
10. Using Inexpensive CMOS Cameras with StrongARM
   Matt Podolsky
11. The Connectivity server
   Suresh Venkatasubramanian
12. Performance Comparison of Alpha and Intel NT Systems using DCPI
   Xiaolan (Catherine) Zhang

Data Cache Optimizations for Java Programs

Nawaaz Ahmed, Cornell University

I interned at SRC from May 19th to August 26th during the summer of 1997. My host for the period of the internship was Mark Vandevoorde.

At Cornell, (where I'm doing my PhD), I work with Prof. Keshav Pingali on compilers for numerical applications. We are working on a different way of looking at programs that is "data-centric" -- since we are concerned about how data is being produced and consumed, we are looking at ways of directly reasoning about data flow instead of manipulating loops as is done currently. The architectures we are concentrating on are distributed shared memory systems (both software and hardware).

I have spent the summer here at SRC exploring methods to reduce data cache stalls. My project was to measure how effectively the caches are being used, to evaluate the potential benefits of cache-related optimizations (e.g., restructuring data), and to suggest how one might perform such optimizations automatically.

To put the work in perspective, at the beginning of the summer the DCPI (Digital Continuous Profiling Infrastructure) group at SRC was looking towards using the data collected by its profiling tools to drive optimizations. DCPI profiles of various benchmarks, e.g., SPEC95, indicate that a significant portion of execution time is spent in stalls due to data cache misses.

As a result, Mark and I developed a methodology for automating optimizations by searching for access patterns in program execution traces. I implemented a tracer that collects information required to suggest the optimizations as well as implementing a cache-simulator which we used to simulate optimizations.

Overview of Specifics

Early on, I decided to focus on Java programs for several reasons. First, the Java Virtual Machine (JVM) makes no guarantees to programmers about data layout, so it is easier to restructure data in Java than in languages like C. Second, to study a program's access patterns, I wanted accurate type information for each instruction that reads or writes data. Java codes (.class files) contain this information. For example the GETFIELD bytecode, which reads a field of an object, identifies the class and the field of the object being read. In principle, C executables could contain similar information in the symbol table section. In practice, with the C compilers that were available to us, rendered the information inaccurate or nonexistent.

Cache Simulator

To measure how well the caches were being used, I converted Sanjay Ghemawat's Java runtime system into a cache simulator that allows for the creation of a multi-level memory hierarchy. Each level of the memory hierarchy is parameterised by the size of the cache at that level, the associativity, the size of the cache-line, and whether allocation is done on read or write accesses. To simplify the simulator, transfer of data between the various memory levels is modeled as being instantaneous. Also, because the JVM is a stack machine, the simulator ignores all stack references rather than simulating stack operations that are unnecessary when registers are

available.

For each cache, the simulator reports the number of read and write hits and misses for the whole cache, for each class, for each field of each class, and for each array type. Thus, one can easily identify the data structures accounting for the majority of cache hits and misses. The cache simulator also keeps track of the fraction of each cache-line that is actually accessed before the cache-line is evicted. This allows us to measure cache-pollution, i.e., the amount of useless data brought into the cache.

I ran the simulator on a suite consisting of three programs:

- javac -- a compiler for java programs
- javacup -- a lalr parser generator
- juno -- a non-linear constraint solver

Using these as benchmarks on an EV5-like cache hierarchy, 50-60% of the evictions take place at the first level cache while only 25% of the cache-line was used. The average number of bytes accessed turns out to be around 13, which represents a cache utilization of just 41%. These numbers get worse for an EV6 like architecture (first level cache line size of 64 bytes vs. 32 bytes on EV5) -- the effective cache-line size is only 21 bytes (33%). Thus as the size of the cache-line increases, optimizations that reduce cache-pollution become more important.

Trace-Driven Optimization

Once the simulator was working, I focussed on the problem of how one might automate memory-related optimizations such as restructuring data to improve locality and prefetching data to mask latencies. A key problem seems to be how to identify common access patterns in a program. Given the common access patterns, one can then attempt to either restructure data to improve locality, or to insert prefetches to mask cache miss latencies.

I built a tool, called tracer, that collects information about access patterns in a program. The information from the tracer can be used to drive several optimizations. I evaluated two optimizations by simulating them in the cache simulator to observe the reduction in the number of read misses. Unfortunately, with only javac and a JIT compiler available, the quality of the code was sufficiently poor that trying to measure a speedup was not feasible: the lack of optimizations like common subexpression elimination, lifting code out of loops, etc. meant that the overhead of data stalls was somewhat dwarfed by other inefficiencies.

Reordering Fields

The first optimization was intended to reorder the fields of a class so that the more frequently accessed fields are clustered together. The goal is to pack frequently accessed data into fewer cache lines.

Reordering did not help much in reducing the number of read misses for our benchmark suite. The reductions in misses ranged from 12% (juno) to 2% (javac) for 32-byte cache lines. For 64-byte cache lines, there was only about a 2% reduction in the number of misses. The reason is that the benchmarks had few objects that were more than 64 bytes long.

Prefetching

The second optimization was to prefetch the target of a handle at the point where the handle is obtained.

For our three benchmark programs using the EV5 cache model, the heuristic prefetched 34% of the first-level misses (introducing an overhead of 6% more load instructions) for javacup; 14.6% (overhead: 3.8%) for javac; and 4.9% (overhead : 0.4%) for juno. Juno did not do particularly well because we did not include uniform array stride access prefetch in our suggestions. Prefetching performed similarly with the EV6 cache model.

Challenges and Future Directions

I found the project very interesting. In my mind it started a chain of speculation about the fundamental issues involved in memory systems. While a lot of work has gone into studying control and data flow, I know of very little work that actually talks about the control flow pattern of the data flow. I think this issue needs to be studied if we wish to understand what is ailing memory systems and how to make them more efficient.

Appendix

```
   JAVACUP

Simulated readmisses on ev5

               L1       L2      L3
        base 5302668  790216  268644      Actual count
   reordered 5052562  791017  267562
  prefetched 4006014  623427  268953
        both 3741982  621708  268177

        base  1.0000  1.0000  1.0000      Percent of base
   reordered  0.9528  1.0010  0.9960
  prefetched  0.7555  0.7889  1.0012
        both  0.7057  0.7868  0.9983

Simulated on ev6

               L1       L2
        base 2836101  185490       Actual count
   reordered 2798603  186224
  prefetched 1745693  131252
        both 1701881  131720

        base  1.0000  1.0000       Percent of base
   reordered  0.9868  1.0040
  prefetched  0.6155  0.7076
        both  0.6001  0.7101



   JAVAC

Simulated readmisses on ev5

               L1       L2      L3
        base 1420800  240719   49399       Actual count
```

```
   reordered 1395944  231566   47033
  prefetched 1304044  215274   48529
        both 1285798  208995   44960

        base  1.0000  1.0000  1.0000       Percent of base
   reordered  0.9825  0.9620  0.9521
  prefetched  0.9178  0.8943  0.9824
        both  0.9050  0.8682  0.9101

Simulated readmisses on ev6

               L1       L2
        base  773516   35406       Actual count
   reordered  755820   33598
  prefetched  679728   31746
        both  672758   28226

        base  1.0000  1.0000       Percent of base
   reordered  0.9771  0.9489
  prefetched  0.8788  0.8966
        both  0.8697  0.7972



   JUNO

Simulated readmisses on ev5

               L1       L2      L3
        base 2502983   37875    2247       Actual count
   reordered 2210184   37954    2263
  prefetched 2385292   37136    2243
        both 2298945   37091    2279

        base  1.0000  1.0000  1.0000       Percent of base
   reordered  0.8830  1.0021  1.0071
  prefetched  0.9530  0.9805  0.9982
        both  0.9185  0.9793  1.0142

Simulated readmisses on ev6

               L1       L2
        base  550187    1034       Actual count
   reordered  544109     860
  prefetched  522144     983
        both  535400     840

        base  1.0000  1.0000       Percent of base
   reordered  0.9890  0.8317
  prefetched  0.9490  0.9507
        both  0.9731  0.8124
```

Min-Wise Independent Permutations

Moses Charikar, Stanford University

Introduction

I have just completed two years of graduate study in the PhD program in Computer Science at Stanford. I work with Rajeev Motwani. My research interests are in the design and analysis of algorithms, more specifically online algorithms and approximation algorithms. Still in the exploratory phase of my PhD, I have tried to work on several different problems within these sub-fields. Some of the work I have done includes algorithms for online page migration, online load balancing, incremental clustering, approximation algorithms for vehicle routing and constructing Steiner trees in directed graphs.

I chose SRC for my summer internship because it has a strong theory group. I saw my internship as a good opportunity to make connections with theory folk at SRC, possibly continuing to work with them after the summer was over. Also, the problems that my host, Andrei Broder said he wanted to look at sounded interesting to me. He was pretty flexible about what exactly I would spend my summer doing and that appealed to me. I also realised that apart from Andrei, I would be able to interact with other theoreticians here and that was attractive.

Research problem

I studied families of permutations which have a particular property called "Min-Wise Independence".

The problem that I studied arises in determining document similarity for documents on the World Wide Web. A mechanism for determining document similarity based on document content is essential for preventing the AltaVista spider from getting caught in spider traps and eliminating spam submitted to AltaVista through the submit URL button. Reducing the size of the index is very valuable since a 20% reduction would reduce the number of TurboLasers needed by four. Also, the sheer magnitude of data we need to deal with makes speed an absolute necessity. This theoretical problem arises in devising a fast scheme for determining document similarity.

Overview Specifics

About a year ago, researchers at SRC came up with the concept called document shingling and devised a mechanism using permutations to extract constant size samples from shingle sets of documents. The constant size samples could then be compared to estimate document resemblance fairly accurately.

For this scheme to work, samples need small, simple families of permutations which have a property called min-wise independence. For a family F of permutations on {1,2, .. n}, roughly this property means that for any subset of X={x_1,...x_k}, for a permutation s chosen uniformly and at random from F, s(x_i) has probability 1/k of being the minimum of s(X) = {s(x_1),s(x_2),..s(x_k)}. Clearly the set of all permutations is min-wise independent. The question is: Can we come up with smaller families that are min-wise independent?

This basic problem is termed the "exact" problem. This requires that the property hold for all subsets of size up to n. We studied the following variants of the 'exact' problem:

1. Size limited

   This requires that the property hold for sets of size up to some threshold k. (This is because for the web application, n=2^64. However, the maximum size of the subsets for which we need this property is twice the maximum document size which is about 2^18).

2. Approximate

   This requires that for a subset of size k, the probability that an element be the minimum of this set under a random permutation from the family be (1+-epsilon)/k. (Since we are estimating resemblance anyway, we can tolerate a small deviation in the exact property).

We looked at combinations of the Size Limited and Approximate problem. We were also interested in small families of permutations with weighted probability distributions.

We were also interested in the performance of simple families such as linear transformations - ax+b (mod p) and in general, pair-wise independent families. For a set of size k, we wanted to bound the probability that an element of this set was the minimum under the family of linear transformations. We wanted to bound the minimum and maximum of this quantity for a set of size k in the worst case as well as bound the expected values of the min and max in the average case.

We could prove a lower bound of about e^n on the size of exact families. This was complemented with a construction of an exact family of size about 4^n. We proved a lower bound of Omega(sqrt(n)2^n) on the size of weighted exact families and had a non-constructive upper bound of n.2^{n-1}. We gave randomized constructions of approximate families of size n^2/epsilon^2 We also gave randomized constructions of size limited approximate families of size k^2 log n/epsilon^2.

We had explicit constructions of size limited approximate families using previously known constructions of approximately k-wise independent distributions.

We proved several lower bounds for size limited and approximate families. In particular, we proved a lower bound of Omega(k.2^k log n) for the size limited problem. This bound involved the notion of Graph Entropy.

For linear transformations, we could prove that the min probability for a set of size k was no smaller than 1/2k in the worst case. The max probability on the other hand could be as large as Omega(log k/k). In fact for the set {0,1,2,... k-1} we determined that the element O has a probability of 3/pi^2 log k/k (asymptotically) of being the minimum. This result involved playing around with Farey sequences and the Mobius inversion formula. We do not have an upper bound on the max probability but conjecture that it is O(log k/k). In the average case, we can prove better bounds on the max probability. Theoretically, we can prove an upper bound of something like 8.3/k on the expected value of the max probability for a random set of size k. Empirical tests show that in fact, the min and max probability seem to be very strongly concentrated around 1/k. We believe that both of them are {1+-o(1)}/k in the average case.

For some of the results mentioned above, we used Maple extensively to facilitate the manipulation of formulae and plot graphs.

The family of linear transformations is pretty close to the family that is actually used in the implementation of the

document similarity testing. The fact that the linear family is pretty good from the min-wise independence point of view (in the average case) can be viewed as some sort of theoretical guarantee that the implementation is sound, that is, the family of permutations it uses satisfies the property that it is required to satisfy.

The Fun and Challenging Parts

A lot of interesting mathematical problems came up in the course of the project. We employed several interesting (and sometimes exotic) mathematical tools to answer the questions that arose - drawing from number theory, graph theory, combinatorics and probability theory. It was a learning experience. I certainly widened my repertoire of mathematical tools. All the questions were challenging as nobody had looked at this problem before and we did not have any prior results to go by. It was definitely a lot of fun being able to answer (at least most of) the questions which we encountered.

Concluding Observations

The most important outcome of my internship was that I made contacts with researchers at SRC, I think the summer internship was immensely fruitful as it laid the foundation for future collaborations with researchers which will undoubtedly be beneficial for me and contribute to my research at Stanford.

I was also impressed by the amount of interaction between the theory folk and systems people at SRC. Such a high level of interaction benefits both the theory and systems communities - massaging good theoretical ideas into something that is implementable and works well, as well as abstracting interesting theoretical questions from problems that arise in practice proved . It is really commendable how the theory people consult extensively on systems projects and still manage to maintain an active theory side by working on pure research problems. I would strive to achieve a good healthy balance of this sort.

Control Flow Graphs for Java Bytecode

Neal Glew, Cornell University

About Myself

I was born in New Zealand and earned a BSc in Mathematics and Computer Science in 1992 and a BSc(Hons) first class in Computer Science in 1993, both from Victoria University of Wellington. In 1994 I travelled to the US for the first time and began my PhD program in computer science at Cornell University. My Cornell advisor is Dexter Kozen whose interests are mainly in theory: algorithms, decidability & complexity of various logics. I also work with Greg Morrisett whose interests are in language theory and implementation. I work on low level type systems. The goal is to produce a language that is close to the machine but probably machine independent yet strongly statically typed with a type safety theorem. This language would be used as the target for translation of a number of high level source languages. This is motivated by the desire to build type-directed compilers and to build secure code download systems. I choose SRC for an internship because of its strength and research interests as well as its West Coast location.

SRC Research

The original project was to look at register allocation in Sanjay's Ghamawat's Java JIT. Unfortunately there wasn't enough infrastructure in place to dive into this project so we started to build one and found it to be interesting in itself. I ended up looking at control flow graphs for Java bytecodes and their uses. This forms a part of improving the overall quality of code generated by jrun. The main problems were how to deal with exceptions and subroutines. We also looked at how to get larger blocks over which a local allocator can work and devised the idea of a superblock. We looked at register allocation and had some ideas. However, time constraints prevented us from exploring further. We did however implement a simple change, that of not saving dead variables at basic block boundaries. The final problem we looked at was how to encode the CFG as a hint in a Java classfile so that a JIT can quickly build and verify the CFG. We managed to devise a scheme that is quick but which is not verifiable and understand some of this tradeoff.

Specifics

I implemented or helped with: A procedure to compute control flow graphs; a procedure to compute superblocks; a procedure to compute live variables; a new code generation infrastructure to support new register allocation schemes; a simple improvement to the present allocator; an experiment with another register allocation idea; and a procedure to compute the sizes of a CFG hint in java class files. This was implemented in C as part of Sanjay's jruntime. I also used DCPI to successfully investigate some performance problems.

Other

I found my project fun and challenging, in particular thinking about how to do register allocation, and the difficulties in making CFGs.

I learnt a lot about Digital, about SRC, and about industrial research labs. I also learnt alot about the research done at SRC in the past through reading many of the SRC research reports. My project taught me a lot about how to measure performance and think about how to improve code quality and measure such improvements.

Extended Static Checking of programs with cyclic dependencies

Rajeev Joshi, University of Texas at Austin

This page describes a summer project undertaken by Rajeev Joshi (UT Austin) while a summer intern at SRC, summer 1997, hosted by Rustan Leino.

Overview

The goal of Extended Static Checking (ESC) is to check, at compile time, for common runtime errors, e.g., dereferencing a `nil` pointer, accessing an array out of bounds. Given a program annotated with a specification, an ESC compiler produces a *verification condition*, a predicate in first-order logic, which expresses that the program meets the given specification. This verification condition is then given to a theorem prover, which checks whether or not it is valid. In most cases, the prover is expected to report either that the condition is valid or that it is invalid (and then also provide an error context); in some cases, the prover may just give up.

My project this summer (under Rustan Leino) was to study some of the difficulties that arise in reasoning about programs with recursive data structures such as linked lists. To track our progress, we chose an example based on the Sieve of Eratosthenes algorithm for generating primes; the example seems to be fairly representative of the class of programs we were interested in. Over the course of the summer, we manually sketched a proof of correctness of this program and determined that we needed to modify one of the original axioms, as the prover is not equipped with induction. I also extended Rustan's ESC/Ecstatic compiler to generate appropriate verification conditions for programs with data abstraction and simple recursive data types. We then tried running the prover on the resulting conditions and found that, with the exception of one axiom schema which we had to instantiate ourselves, the entire example could be checked automatically.

In the remainder of this note, I shall give a brief introduction to ESC and describe the problem that I worked on during the summer.

Background

Consider the following sample specification:

```
class Rational
{       field num, den : integer
   spec field ok : boolean = (den # 0)
}

procedure normalise(r : Rational)
{   pre       r.ok
    modifies  r.num, r.den
    post      r.num * r.den' = r.num' * r.den
}
```

Class `Rational` has three fields: two integer fields representing the numerator and the denominator, and a *specification field*, whose value is a function of the values of other fields. In ESC jargon, the specification field is said to *depend* on the fields of which it is a function. In the example above, for all objects `r` of type `Rational`, we have that `r.ok` depends on `r.den` .

The specification of procedure `normalise` has three clauses which refer, in order, to the precondition, the modification list, and the postcondition respectively. The precondition is a first-order predicate which is assumed to hold upon entry into the procedure; it is expected to be established by the caller. The postcondition is a first-order predicate which is required to hold upon exit from the procedure body; it relates unprimed variables (denoting variable values upon entry) and primed variables (denoting variable values upon exit) and it is expected to be satisfied by the procedure implementation. The modification list describes the variables that the implementation is allowed to modify; it is translated into predicates, called *modification constraints*, which are conjoined to the postcondition.

Informally speaking, a modification constraint asserts that the value of a field (such as `den` above) changes only in those objects where the procedure is allowed to modify it. Modification lists may mention specification fields; in such cases, the list is first *desugared* (i.e., rewritten) into another list before modification constraints are generated. The rules for desugaring modification lists are somewhat complex -- they are based on engineering decisions intended to make them easier for programmers to use -- but, to get some idea of how they work, consider the modification list

```
modifies r.ok
```

for the example above. This list would be desugared into

```
modifies r.ok , r.den
```

which produces the following modification constraints, where `s` ranges over objects of type `Rational`:

```
(All s :: s.ok = s.ok'    \/  s = r)
(All s :: s.den = s.den'  \/  s = r)
(All s :: s.num = s.num')
```

In more complex examples in which several levels of abstraction are layered one on top of another, desugaring a modification list typically involves computing a transitive closure of the dependency relation.

Cyclic Dependencies

To understand the difficulties that may arise with recursive data structures, consider the following definition of a linked list:

```
class List
{       field value : integer
        field next : List
   spec field valid : boolean = (value # 0  /\  (next = nil \/ next.valid))
}

procedure Init(l : List)
{   modifies  l.valid
    post      l.valid'
```

```
}
```

Note that the declaration of field `valid` introduces the following dependency:

> for all objects `l` of type `List` , `l.valid` depends on `l.next.valid` .

Such a dependency of a specification field (`valid` at object `l`) on itself (`valid` at object `l.next`) is called a *cyclic dependency*. A preliminary difficulty with cyclic dependencies is finding a compact representation for modification lists. Recall that desugaring typically requires computing the transitive closure of the dependency relation. In the presence of cyclic dependencies, this closure is no longer finite. For instance, a naive desugaring of the modification list for procedure `Init` above yields

```
modifies l.valid, l.next.valid, l.next.next.valid, ...
         l.value, l.next.value, l.next.next.value, ...
         l.next, l.next.next, l.next.next.next, ...
```

so we need a way to represent such infinite sets succinctly. A simple solution is to use a closure operator "*" (pronounced "star") which denotes "0 or more occurrences of" ; then, the modification list above may be written as

```
modifies l.next*.valid, l.next*.value, l.next*.next .
```

The next difficulty is to determine how to generate appropriate modification constaints for such expressions. A naive translation of

```
modifies l.next*.value
```

would give

```
(All s :: s.value = s.value'  \/  (s \in l.next*)) .
```

Since the prover is not equipped to reason about such closures, we rewrite the second disjunct using a 4-argument predicate `REACH`, introduced in 1983 by Greg Nelson (who also provided a set of useful axioms for it). Informally speaking, `(REACH u z f x)` asserts that it is possible to reach object `z` from object `u` by following the pointer field `f`, without ever following the `f` pointer of object `x`. For the example above, the modification constraint would be written as

```
(All s :: s.value = s.value'  \/  (REACH l s next nil) ) .
```

Some Discoveries

In proving the Sieve of Eratosthenes example, we found that we needed only two axioms about `REACH` (viz., A1 and A2 from Nelson's paper). We also discovered that we had to modify the *pointwiseness axiom* schema for cyclic dependencies (due to Dave Detlefs), which, for the example above, yielded

```
(All s ::
   (All w : w # nil  /\  (REACH s w next nil)
          : w.value = w.value'  /\   w.next = w.next')
==>
   s.valid = s.valid' )
```

We found that, since the prover is not equipped for induction, this axiom was not enough to prove even the following simple Hoare triple (due to Jim Saxe):

```
      { s.valid  /\  s # t  /\  (REACH s t next nil)  /\  t.valid }
    t.next := nil
      { s.valid }
```

Our new axiom schema now yields

```
    (All s ::
       (All w : w # nil  /\  (REACH s w next nil)
              :    (w.value = w.value'  /\  w.next = w.next')
                \/ w.valid = w.valid' )
    ==>
       s.valid = s.valid' )
```

which seems sufficient, at least for the examples we have considered.

Future Work

There are at least two directions for future work. The first is to study programs with recursive data structures in which more than one field is involved in the recursion. (An example is a binary tree, with pointers for left- and right- children.) This involves generalising the `REACH` predicate to accept a relation (or a set of fields) as its third argument. Since we have used only 2 axioms about `REACH` so far, this extension may turn out to be reasonably straightforward. The second direction of research is to find sufficient conditions under which users may not introduce inconsistencies into the checker. Currently, as pointed out by Jim Saxe, it is possible to introduce an inconsistency by using cyclic dependencies. As an example, consider changing the definition of `valid` to

```
    spec field valid : boolean = (value # 0  /\  (next = nil  \/
~next.valid))
```

where ~ denotes negation. Suppose initially that there is only one object `t`, whose `value` field is nonzero and whose `next` field is `nil`. Then the assignment

```
    t.next := t
```

introduces an inconsistency, since now there is no solution to the recursive equation in `valid`.

WebL--Searching in Structured Text

Thomas Kistler, University of California at Irvine

I am a third year Ph.D. student at the University of California in Irvine, working with Michael Franz on a new operating system that reconciles the advantages of portable executables with high performance. For portable executables, most of the traditional optimizations cannot be performed at compile time any longer, resulting in serious performance problems. This flaw can only be overcome by delaying optimization until runtime (or load time) and enriching current operating systems with runtime optimization infrastructures. Our new operating system implements such an infrastructure and continuously and gradually optimizes programs in the background or on specific request of the programmer. It also utilizes an adaptive profiler to custom-tailor optimizations towards system- and user-behavior.

At SRC this summer I have been working with Hannes Marais on the implementation of WebL. WebL is a new Web scripting language that has been designed to automate tasks such as retrieving Web documents, extracting structured and unstructured data from Web documents (for example HTML-, and XML-based Web pages), and creating and manipulating Web documents. One of the main problems that we were trying to solve this summer was the design of an expressive, powerful and concise query language for WebL that allows searching on the structure and on the flat text of HTML- or XML-pages. Traditionally, approaches for searching in structured text documents have focused on either searching documents by content (e.g. searching with regular expressions) or by structure (e.g. subtree matching or context-free grammar matching), but not both at the same time. Most of the query languages also lack orthogonality and compositionality, and, in most cases, do not allow the expression of overlapping of search results. To overcome these problems, we developed a novel combined approach for searching in Web documents that allows mixing content and structure in a simple, orthogonal, and concise query language. Rather than being based on trees or grammars, it is based on set algebra. The basic components of the proposed search algebra are sets of pieces and set-operators. A piece is a continuous region in the text that can either be constructed by searching the text with a regular expression or searching for markup elements in the structure. Set operators allow combining piece-sets to construct more powerful queries. WebL currently supports a combination of structural, positional, and basic set operators (e.g. in, contain, after, before, union, etc.).

One of the most important things that I learnt this summer is that the design of a programming language is much more difficult than it would appear when reading the final language specification. Finding the essence of the problem and the right mix between simplicity and expressiveness is a tedious process that involves a lot of discussion and often requires starting from scratch again and discarding previous ideas and implementations.

Generating Juno-2 Figures as Java Bytecodes

James Leifer, Cambridge University

I am two years into my Ph.D. at the Cambridge University Computing Lab (in the UK) where I work with Robin Milner on process algebras (pi-calculus and action calculi). My work is funded by the U.S. National Science Foundation and the British Goverment. I did an undergraduate degree at Oxford ("the Other Place" as they say at Cambridge) where I worked under Bernard Sufrin and C. A. R Hoare. Despite all that, I'm a New Yorker (or at least can pretend to be one).

I decided to apply for a SRC internship because of SRC's reputation for combining systems and theory work, which suited my interests exactly. From Cambridge, I discussed possible projects with Greg Nelson and we agreed to do work involving some "honest programming" - - -- something that I had not done very much of while pursuing my Ph.D.

The goal of the "honest programming" was to extend Juno ---a dual-view graphics editor built by Greg Nelson and Allan Heydon at SRC--- to allow it to emit Java applets that render Juno graphics on the Web. Juno has a programming language based on Dijkstra's guarded commands (extended with geometrical constraint solving) and its approach to graphics and animation is to create a program that renders the graphics as a side effect of execution. Internally, Juno compiles programs to a bytecode and runs them on a VM.

This summer I built a compiler that translates a Juno program into a Java class file, and I constructed a library of Java classes that provides run-time support. The compiler translates modules to classes, global variables and constants to fields, and procedures to methods. The compiler handles almost all Juno constructions, including modules and imports, global variables and constants, procedures, functions, predicates, control structures, and expressions. The run-time system provides mathematics and PostScript drawing support. Both the compiler and run-time system do not yet handle closures and constraint solving, nor do the graphics libraries do double-buffering, which is necessary for flicker-free animation, but static graphics are working well.

There were several differences between Juno and Java which influenced the design of the translator.

1. Juno programs are dynamically typed ---values range over numbers, strings, pairs, etc., and this was mirrored in Java by creating a class JV (for "Juno value") and subclasses of JV corresponding to Juno's value types.

2. Juno's procedures can return multiple out-parameters but Java's methods cannot. In our implementation a method returns a reference to an array containing the out-parameters.

3. In Juno certain expressions are not strict, that is, they are defined even when subexpressions are not.

This doesn't fit well with Java's notion of exceptions, because when a Java exception is thrown, the entire stack is deleted, not just the items placed on the stack in the surrounding try-catch block. We got around this by creating a global bit that is set as certain parts of an expression are evaluated and tested so as to obtain the correct semantics.

This project was the largest engineering artifact I have ever constructed and I learned much from working with Greg and Allan on how to structure a complicated system and how to test it. I also had many lively talks with others in the lab who gave me good advice and discussed with me their research interests. And, of course, I thoroughly enjoyed the California weather and food!

A High speed Interface from PCI to FireWire

Oskar Mencer, Stanford University

About Me

I'm a third year PhD student at the Department of Electrical Engineering in Stanford. I'm working with Mike Flynn and Martin Morf in the Computer Architecture and Arithmetic Group. Currently I am working with the PCI Pamette on various aspects of adaptive computing. Before joining Stanford, I earned an undergraduate degree at the Technion - the Israel Institute of Technology. I chose SRC for my internship because I think that SRC is one of the most interresting places for research in computer systems.

Summary of the Project at SRC

We designed a high-speed interface between PCI and the Link Layer of the IEEE 1394 FireWire chipset from Texas Instruments. Our environment allows us to exercise all the various features of the FireLink chipset from Texas Instruments. Highlights of the design are: PCI write bursts from CPU to PCI, DMA back to host memory, and flexible buffers to deal with variable latency Link Layer chips. The interface is implemented on the PCI Pamette FPGA board.

FireWire: the Interconnect for Multimedia and more

FireWire, IEEE standard 1394, is a serial interconnect developed by Apple and Texas Instruments. The standard regulates the physical and link layer. The primary target applications are Audio Video and today's SCSI connections. The highlights are: 200 Mbits/s of physical bandwidth, user-friendly hot plugging, low cost and it's a non-proprietary environment. Didier Roncin developed a FireWire daughter board for the DEC PCI Pamette, called FireLink, consisting of two FireWire Channels and one Xilinx XC4010E for control. Strict compliance with the FireWire standard and flexible FPGA technology make this board a useful platform for exploring FireWire.

The FireLink Library

The software interface to FireLink is also designed with performance as the main target. Therefore we chose the C programming language and macros for communication with the FireLink hardware, similar to ANL macros. The library implements a message passing abstraction on to the FireWire.

Performance

Performance is analyzed for three parts of the FireWire network. "Send", the time to write a burst-block of a specific size into the transmit FIFO, has a maximal bandwidth of 70 Mbits/s for block sizes of 240 quadpackets. ReadAck, or the time for the link layer to send all the data from the transmit FIFOs to the destination and receive an acknowledgement from the other side, translates into a bandwidth of 140 Mbits/s. (According to the documentation from Texas Instruments, the physical layer transmits data at 200 Mbits/s.) Recv, or the time to

move the data from the receive FIFO at the link layer chips to main memory, results in a bandwidth of 110 Mbits/s. This is achieved by DMA bursts from the PCI Pamette board to main memory.

Future Work

One possible extension to this project is to create clusters of machines, examine the performance and compare with other competing interconnection technologies. In addition, a software interface to Memory Channel technology would integrate all the Memory Channel applications with FireWire. A more esoteric project would be to think of the FireLink hardware as a distributed system of custom computing machines (Pamette's). The objective could be to speed up computation intensive parallel applications by accelerating each node with custom FPGA designs.

What I learned from my internship

I strongly improved my FPGA design skills. As a side project I also implemented a Java interface which talked to the PCI Pamette through an HTTP link and a dedicated Tcl Webserver. In conclusion, DEC SRC has met all my expectations and more.

SPHINX: Site-specific Processors for Html INformation eXtraction

Robert C. Miller, Carnegie Mellon University

Introduction

I'm a second-year CS PhD student at Carnegie Mellon University, I work with Brad Myers on programming-by-demonstration and the Amulet user interface toolkit. My recent research at CMU has involved animations in Amulet, and applying programming-by-demonstration to the World Wide Web. I developed a demonstrational system that infers, from a single demonstration, how to construct a "composite Web page" by extracting pieces from other Web pages and combining them.

I chose SRC for my internship this summer over Xerox PARC and FXPAL (both of which made me offers) mainly because I've talked to other grad students who had very positive summer experiences at SRC, and the SRC project was most closely aligned with my interests. SRC also responded to my internship application very quickly -- less than two days after I submitted it -- which is a compliment to their organization and preparation.

Crawlers, also called robots or spiders, are programs that traverse and process the World Wide Web automatically. Examples of crawling applications include indexing, link-checking, meta-services (like meta-search engines), and site downloading. It turns out that crawlers are difficult to write for several reasons: (1) a lack of good library support, so even simple crawlers take work; (2) multithreading is required for good performance; and (3) writing site-specific crawlers (like meta-searchers) takes much trial-and-error to figure out which links to crawl and how to parse pages, and then the Web site's format changes and the work must be thrown away.

My host for this work was Krishna Bharat.

Solution

We built a system called SPHINX (Site-specific Processors for Html INformation eXtraction). SPHINX is a user interface and Java class library that supports developing and running crawlers from a user's Web browser. For users, the SPHINX user interface offers a number of advantages. One advantage is that common crawling operations can be specified interactively, such as saving, printing, or extracting data from multiple Web pages, which makes simple crawlers simple to write. Another is that the pages and links explored by the crawler can be displayed in several visualizations, including a graph view and an outline view. For Java programmers writing custom crawlers, the SPHINX library provides multithreaded crawling, HTML parsing, pattern matching, and Web visualization, along with the ability to configure and run the custom crawler in the SPHINX user interface.

Observations and Future Work

The SPHINX user interface runs as a Java applet hosted by a Web browser. This decision to run SPHINX inside a Web browser, as a privileged Java applet, turned out to be an effective strategy. As a consequence, SPHINX crawlers are portable, require no special configuration to access the Web, and see the Web exactly as

the user sees it. In particular, when a SPHINX crawler requests a page, it uses the same cache, authentication, proxy, cookies, and user-agent as the user, ensuring that it gets the same response back from the server that the user would.

For future work, it would be interesting to scale the SPHINX architecture to Web-wide crawlers, such as search engine indexing or Web archiving, which retrieve and process millions of pages. Such crawlers typically run on a farm of workstations, raising interesting issues such as how to divide the crawling workload fairly and how much information must be shared by cooperating crawlers. Also, the platform-independence and safety of Java imply that SPHINX crawlers could be moved around the network easily, to access the Web at the most convenient point. Exploring the architecture and security policies of server-side crawlers would be an interesting direction for future work.

Vis: A Unified Graphical User Interface for DCPI

Robert O'Callahan, Carnegie Mellon University

Background

I'm a third year PhD student at Carnegie Mellon. My work is primarily in large-scale program analysis tools. I chose SRC because I thought it would be a fun place to spend a summer and it has many interesting people and projects. I was looking forward to getting experience working as part of a group, and at doing something a little different from my main line of work.

The Project

The Digital Continuous Profiling Infrastructure (DCPI) is a suite of tools that performs sample-based profiling with very low overhead. The profiles cover the entire system, including the kernel, and the system is very convenient to set up and operates transparently. There are many tools for reviewing the data, including tools that estimate execution frequencies, average instruction stall times and reasons for those stalls. Thus DCPI gives the user access to a lot of information at the level of entire binaries right down to individual instructions.

One problem with the system is that there are many distinct tools used to view and interpret the data, and these tools are not integrated. Furthermore, they are mostly text-based, which limits the ways information can be displayed, and they mostly do not give the user ways to interact with the data. Therefore I was given the job of creating a graphical user interface that would:

1. subsume the displays of most of the tools
2. use graphics to increase the density of information display and make the interface easier to use
3. provide interaction and integration so that all features would be easily accessible.

Approach

I created a Java application that communicates with the DCPI libraries using HTTP. The libraries are packaged up into a CGI server that answers queries sent by the application. The resulting Java application can be run on any JVM, including inside a Web browser, and can view profile data from any machine on the network.

For the interface itself, I used Scott Hudson's Subarctic toolkit. This is basically a widget library on top of AWT that provides easy handling of common input behaviours such as dragging, a constraint system that provides powerful ways to position UI elements dynamically, and some other useful hacks such as double-buffering.

When designing the interface, we focused on a few design principles. One was consistency. We made globally consistent choices of colors for different elements (e.g. cycle counts). Globally choosing scales is impractical because the scales of instruction counts can be orders of magnitude different from the scales of counts for entire programs, so we chose scales to be consistent within windows but not necessarily between windows. Of course,

we followed standard guidelines in trying to make all the UI widgets behave in similar (and generally familiar) ways.

Another principle was configurability. Colors, fonts, sizes and orientations of almost everything in the interface can be easily modified by the user, often by directly manipulating the interface itself using the mouse.

Another principle was economy. The display is organised as a set of views of five different kinds (lists of binary images, lists of procedures, lists of basic blocks, listings of entire procedures, and control-flow graphs of entire procedures). Each view displays a limited amount of data about just the objects in that view. More details about objects in the view can be obtained by holding the mouse cursor steady over an object; a temporary "tip" will pop up to display the information. Furthermore, when the user clicks on an object in one view to highlight it, other views will automatically scroll to and highlight any related objects. Thus the multiple views work together to display more information.

Results

It was fun to build the system and try out different ways of displaying the data. We learned that simple displays were often the best fit for people's needs. While complex visualizations could have looked prettier, in this case they don't seem to be required.

Subarctic encourages the programmer to use the "structured graphics" model, where each graphical element has an underlying widget. Thus, in applications such as Vis that can have thousands of data items to display, the number of widgets can become very large. This can easily lead to performance problems due to the large amount of space and time required to manage these data structures. In Vis, these problems are barely manageable. An interesting question (outside the scope of my work) is whether there is a way to maintain the ease of programming and code reuse that derive from the structured graphics model while eliminating the need for huge data structures that cause the performance problems.

In order to get the tool to work acceptably on large data sets, we had to do a fair amount of tuning. It turned out that different Java VMs have greatly different performance characteristics; relative speeds vary a lot depending on the kind of code being executed. In other words, in addition to the architectural innovations that make performance ever harder to understand, we now have an extra software layer that also adds to the problem, and the number of architecture/VM combinations is significantly larger than the number of architectures.

Future Work

Some of the lower-level lessons I learned, such as how large Java programs are structured and how their performance can behave, will help me in my work at CMU as I apply my program analysis techniques to Java. I hope that the Vis tool itself will be used by the DCPI group and its users. As DCPI evolves, the tool will need to be updated; in particular, when new kinds of information become available, new views may have to be created.

Using Inexpensive CMOS Cameras with StrongARM

Matt Podolsky, University of California at Berkeley

I've just finished my third year of graduate school at U.C. Berkeley, where I study electrical engineering. I work with Martin Vetterli and Steve McCanne, and my general interests concern signal processing and networks. Specifically, I've been studying the use of Forward Error Correction for real-time audio over the Internet. Various Internet audio tools (e.g. rat, freephone) have added extra redundancy to their audio streams to protect against packet loss. I'm interested in trying to determine how much redundancy to add, and also in determining how things change as network conditions (e.g. traffic distributions) change. In the near future I'll be studying similar issues with video.

While at SRC, I worked on attaching small cameras to SHARKs (the Digital Network Appliance Reference Design). The sensors for these cameras are made from a standard CMOS process (as opposed to a traditional CCD sensor), and so they can be fully integrated with other logic on a single board. Other advantages over CCD systems include low price (only $15-$20 for one of these camera sensors and a plastic lens, when purchased in quantity) and low power (between 250 and 375 milliwatts). Mark Hayter and I originally envisioned my project as studying image and video processing algorithms on the StrongARM microprocessor using a SHARK connected to one of these cameras. However, when I started work this summer much needed to be done before the camera could send video to a SHARK. As such, my summer was spent working on the following elements:

1. Writing device drivers for two different CMOS sensors. The camera sensors were connected to a rev 1 SHARK through its ROM card slot. A board consisting of an ADC, a pair of dual FIFOs, and a PAL acted as a bridge between the sensor and the SHARK' ROM card. Device drivers were written in C, and accessed the camera and FIFOs through memory mapped addresses on the SHARK. The camera sensors' configuration data is read and written via an I2C serial bus interface; the driver I wrote included code to handle this serial communication.

2. Porting the UCB multicast tools to SHARK. I ported UCB MASH multicast toolkit and got the shared media board (mb) and video conferencing tool (vic) to run on a SHARK. With vic you can display refreshing video locally and also multicast it to other workstations. Because simply displaying the video on the local machine with the camera still involves doing an encode and decode for each frame, the video frame rate was only about two frames per second (fps).

3. Adapted a simpler Tcl/Tk display program for displaying video. Using a program Mark Hayter had written for displaying video from one of these sensors on a Lectrice, a tablet-based computer, I modified it to work on SHARKS with my device driver. Its simplicity allowed me to study where time was being spent to process and display video, as well as display video faster than vic (by the end of the summer we had about 4 fps using this program). I also adapted Tk widgets to controls various camera parameters, like ADC gain, black calibration levels, and color balance.

4. Improving picture quality. The initial video out of the analog VV5426 camera sensor we used was quite noisy. The noise came from the power supply interfering with the ADC, and this was solved by moving on

to the VV5404 digital sensor. This one took advantage of the CMOS nature of the sensor to put an ADC right on the board with the sensor. The resulting picture was noise free and extremely sharp. However, the color was rather unattractive, and using the Tcl/Tk program I was able to determine that the red and blue levels needed to be boosted, while the green levels should be reduced. Other parameters like black levels and gain were varied and used to further improve picture quality.

5. Performance optimizations. Finally, I was able to study the performance of the camera. One of the areas I focused on was the color extraction process. The CMOS camera sensors consist of an array of alternating color filters (one line has RGRGRG..., while the next has GBGBGB..., and back to RGRGRG, where R=red, G=green, and B=blue) in front of light sensors. In order to obtain a full color picture, the software supplied with the cameras uses a Bayer algorithm to interpolate the missing colors. However, even though you start off with one-third of the color information you need (one half of the green and one-quarter each of the red and blue), the full-color output image is offset from the camera input image by one-half a pixel in the X and Y directions, so that every output pixel requires that all three colors be averaged. I implemented a simpler symmetric filter which uses only shifts and adds, and reduced the color interpolation time by 33%. I also saved time in the next step of image extraction, a high-frequency emphasis filter, by removing some unnecessary thresholding checks and hard-coding in color gains.

The most fun part of this project was getting the digital sensor working with corrected color and seeing good quality images come into the SHARK through a $20 camera. One of the most challenging aspects of the project was working with the sensors, for I was dealing with extremely limited documentation. This meant that I had to reverse engineer the vendor software and some of their data modes in order to determine functionality. It also meant finding and debugging inaccuracies in supplied data sheets. Dealing with SHARKs was also rewarding and fun, because their capabilities and state of development frequently amazed me, but also challenging because they were still prototypes. All in all I found this to be a very rewarding project, and a great opportunity to work at SRC and interact with so many talented and capable people.

The Connectivity Server

Suresh Venkatasubramanian, Stanford University

I'm a third year Ph.D student in Computer Science at Stanford University. At Stanford, I work with Rajeev Motwani and Jean-Claude Latombe on problems in geometric pattern matching (with application to drug design). In general, I'm interested in theoretical work as applied to real-world problems, and within that framework, the project that was proposed to me at SRC sounded interesting.

The problem is the following: Using link information on web pages, construct a database that maintains the graph of web pages (and the edges between them) and answer connectivity queries about this structure. This information is useful for a variety of reasons. One of the most important is as a filter for ranking algorithms that use connectivity information to determine the relevance of query results.

What I mainly worked on was designing and implementing the architecture for this server. Some of the design issues were the following:

1. Resources: We use a 4GB machine with a large disk, so we can handle large amounts of main memory storage.
2. Updates: We chose an update model that is batched (due to the fact that we obtain updates from the AltaVista crawler once a day).
3. Querying flexibility: We keep the API simple so as to allow a variety of search mechanisms to be built on top of it.
4. Communication with clients: Our current interface is HTTP based, via a custom HTTP server - again, this is flexible.

The graph structure is relatively straightforward. We maintain a record for each node containing a list of its inlinks and outlinks. In the actual implementation, this list is actually a pointer into a table of all the (in/out)lists. The nodes are represented not as URLs, but using an internal node ID.

Designing the URL database turned out to be the non-trivial part. Initially, we were using the Altavista-assigned fingerprint as a node ID, and used a huge file containing the ID-URL correspondence to compute the mapping from an ID to a URL. The forward mapping is a function computation. However, this approach was not useful for two reasons:

1. Fingerprints are 64 bit objects, which is unnecessarily large (we only have 250 million URLs right now). This translates into a space wastage both in the URL database and in the graph structure.
2. The list of sorted URLs is over 19 GB. This is very difficult to manage, and we need to compress the data to improve space and lookup.

As a result, we use assigned 32 bit numbers as node IDs, and use an encoding scheme to represent the URLs. This scheme is a delta-encoding scheme, where each string is stored not in its entirety, but as the difference between it and the string before it. This allows us to significantly compress the data - down to about 5.3 GB. Note that each compressed entry also contains the node ID stored within it.

We also build an index to search this database (delta-encoding a file forces the search to be linear, so we need an index containing fully formed URLs which we can use as a start point for decoding the URL).

In addition to building the above structure, we also built a system for updates that takes a formatted Altavista crawl result and merges it into the database. This system is non-trivial, owing to the sizes involved and the need to perform updates quickly.

The interesting part of the work was designing the data structures. We came up with a variety of schemes that could be used to represent the URL database, and even toyed with the idea of changing the graph representation to something more intricate. Ultimately though, and this is probably the main lesson to be drawn, simplicity should not be underrated. Our final structures are quite simple, which means that writing code for them is easy, and performance is generally acceptable. Moreover, a simple structure allows us to (potentially) play with many different types of query families without introducing a bias into performance. The largest single amount of effort however, went into initializing the databases from the latest AV crawl (about a month old).

This work has no real connection to the work I do at Stanford, (which is more theoretical in nature), but I found it interesting mainly because it is a good example of the sort of problem I'm interested in, namely, where there are theoretical issues to explore as well as non-trivial system-building problems. Due to time constraints, I was unable to look at related graph-theoretic work (mostly because of time), but this project gave me a lot of experience in the nitty-gritty of data structure design and implementation.

Acknowledgements: I'd like to thank Puneet Kumar and Andrei Broder for their constant help and support, and for making sure I didn't keep veering off the focus of the project. A special thanks to Hannes Marais for always being around as a "covert" third host. I would also like to thank Krishna Bharat, Mike Burrows, Sanjay Ghemawat, and Monika Henzinger for their help.

Performance Comparison of Alpha and Intel NT Systems using DCPI

Xiaolan (Catherine) Zhang, Harvard University

Introduction

I am a third year Ph.D. student at Harvard University where I work with Brad Chen on the Morph project. Morph is a system that aims to provide a framework for automatically monitoring and optimizing software that runs on an end user's computer. I have developed the Morph Continuous Monitor that monitors the execution of the application by sampling the PC (program counter) of the executable. The samples collected by the Monitor are then fed to a profile-driven optimizer.

I was an intern from August 23 to November 21; Mark Vandevoorde was my host. I chose SRC for internship because of its excellence in computer science research and the high reputation of its intern program.

SRC project

While at SRC, I worked on comparing an Alpha 21164 (Miata) NT system with a PentiumII NT system using the DCPI (Digital Continuous Profiling Infrastructure) tools. The goal was to figure out why some applications run faster on one system than the other and try to look for ways to improve applications running on the Alpha NT system.

Experimental Setup

The two systems we compared were a 500 MHz Alpha 21164 EV56 (Miata) system and a 266MHz PentiumII system. The applications we studied were the BAPCo SYSmark NT4.0, Aladdin ghostscript 5.0.3, and MicroSoft SQLServer 6.5 running the TPC-B benchmark as the workload.

For performance analysis, we used the aggregate event counts collected using DCPI and the DCPI tools that provide instruction level stall analysis (Alpha version only). Ntprof was used to discover unaligned loads. For (manual) optimizations, we used the NT Atom instrumentation tool to perform binary rewriting.

Results

Overall performance

In terms of running times of the benchmarking programs, the performance of the two systems are pretty close except for Ghostscript and PowerPoint. For Ghostscript, the Miata is much faster. PowerPoint is a 16-bit Windows application and is interpreted on Alpha, which results in a much slower running time.

One interesting finding is that the number of PentiumII micro-ops (one Intel CISC instruction is decoded into RISC-like micro-ops before execution) and the number of Alpha RISC instructions are pretty comparable. For

system code, the number of PentiumII micro-ops are consistenly larger by a small fraction.

We also discovered that for the BAPCo SYSmark programs, the instruction cache performance for PentiumII system is much better than the Alpha, which is responsible for a large fraction of the stalls for Excel and Word.

For SQLServer 6.5, both systems performed similarly because the server was not CPU-bound. We also discovered that the Alpha version was compiled for debugging!

Detailed Examples

- Case 1: We identified an unaligned load in ghostscript for Alpha NT that is responsible for 29% of the total cycles, and we worked with Peter Deutsch to fix the problem. The fix will appear in the next release of Alladin Ghostscript.

- Case 2: We identified a misuse of mb instruction in the Alpha mga device driver which is responsible for 16% of the total cycles for ghostscript. A driver that is optimized for Alpha can be downloaded from the Digital Web site which fixes this problem.

- Case 3: A simple prefetching optimization on Texim improves running time by 9%.

The fun and challenging parts

A challenging part of the project was to work with tools that were not yet stable and to provide useful feedback to the authors to help improve the tools. I have learned a great deal about hardware architectures and Windows NT.