Technical report

Dynamic Optimization

by Relocation of Data

by Jean Christophe Beyler

CAPSL - University of Delaware

under the supervision of Professor Gao

ICPS/LSIIT Université Louis-Pasteur de Strasbourg

under the surpervision of Professor Clauss

02-2005/05-2005

Contents

List of Figures

List of Tables

List of Algorithms

Thanks

I would like to take the time to thank Professor Gao and Professor Clauss, both of them made this trip possible. I learnt a lot and it was a privilege to go to the University of Delaware, even if it was for a short while. They helped me throughout my stay, always guiding me the best they could and I tried my best to listen and follow the yellow brick road. I'd like to thank the CAPSL team, everybody was very friendly and I always had interesting discussions when the chance arised.

Finally, I'd like to thank the people I met in Newark, who made the stay fun and sociable. The discussions were not always about computer science or about my work but I also learnt a lot from them, and I am always grateful to have the chance to meet so many good people. They'll probably never read these lines but at least it is written somewhere.

Last but definetely not least, I'll thank Tcla, my fiance, who let me leave her side to make this trip.

Introduction

Dynamic optimization is an emerging field because there is only so much static optimizations can do. Already, profiling schemes or programmer directives are used to help the compiler because, as programs get bigger and more complex, the compiler is having more and more trouble understanding how to optimize the code.

In the word *dynamic*, we will note a sense of *Just-in-time* compilation [2, 10]. The idea is to wait for the execution of the program, try to understand where the bottlenecks are and optimize them. The optimization can be done by modifying the instruction code, the data layout or any other element that is involved in the execution of a program.

Other Dynamic optimizers have already been implemented [11, 4, 6, 9, 8, 12, 3, 14], but none, to our knowledge, have really tried to tackle the problem of reallocating memory. It is true that our methods will use more memory but, as all optimization schemes, we are trying to reduce one aspect of a program, even if this means reducing the performance of another aspect. Thus, we might be using more memory, but since we are looking for a speed-up, it is not always a problem. Once we have a speed-up, we can then decide if the cost in terms of memory use is acceptable for a certain acceleration or not.

The idea, that we will be working with, is that many data accesses sequences cause data-cache misses because the stride incurred is too great compared to the size of a line of cache. One possible solution is to extract the load(s) that cause the cache-misses and store the data in a *data-buffer*. Then when the data is accessed, we will use our own data-buffer where the data has been reorganized. In a certain sense, we will be creating a new kind of buffer. A data-cache ***for*** data-caches, something that we could call a L0 cache (eventhough, our solution is entirely software). We will be working with the *Itanium-2* processor [7] because of it's EPIC infrastructure. The use of bundles is a practical low-level parallelism that we will try to use to achieve an even greater speed-up.

One of the challenges when working in the dynamic field is to be able to find the function that we want to modify. This is done by using the ELF Format [1]. Once the function is found, we will probably have to optimize the code. This means editing the text-segment of a program while it is writing. We will present how this is done.

This technical report will start by explaining those two points before handling two simple examples with two different versions of our model. For the moment, this model is still very simple and not yet totally functionnal but the early results make us feel that there is a good chance that, in the end, we will have a general model able to optimize a lot of different kinds of applications.

Chapter 1

Related work

1.1 Dynamic optimizers

1.1.1 Dynamo

Hewlett-Packard presented in 2000 an optimizer called *Dynamo* (Dynamic Optimisation) [4]. Its principal trait was that it was transparent to the user. The meaning of transparent lies in the fact that the programmer did not have to modify his/her code to use *Dynamo*. The user only had to specify an option while compiling the program to include the optimizer.

To briefly explain what this system does, we will just say that it interprets the code during execution and, when it notices a hot-trace, it will extract the trace, optimize it by creating a new block of code, and use the new version to achieve a speed-up. The new code is put into a shared-memory block.

A technique was put up to *flush* the memory to reduce the total memory needed for the optimizer. The authors based their model on the assumption that a program is divided into a number of phases and, each phase, has it's own behaviour. If *Dynamo* is creating a lot of new versions of code, it concludes that it is entering a new phase. Therefore, the old blocks of code that were created previously will probably not be used anymore. A flush is then made to reduce the memory use.

1.1.2 *Deli*

In 2002, another team from Hewlett-Packard presented a project named *DELI* (Dynamic Execution Layer Interface) [6].

This system had two execution versions:

- A transparent version that makes this model look a lot like *Dynamo*.
- A second version that is actually an API (less than 20 functions) that enables the user to define and modify the basic behaviour of the model.

The API version transforms the optimizer into a more powerful system. It is capable of running on a PocketPc by optimizing directly the Operating System, but also by modifying dynamically the code. Suppose the processor can execute an assembly instruction faster using a specific one that the compiler does not generate. Using the *DELI* model, we can change the instruction using a *just-in-time* technique. The model will check the instructions sent to the processor and switch the instruction that can be optimized. This is only one example of the capabilities of the *DELI* system.

1.1.3 *Adore*

Though the *Dynamo* and *DELI* systems are very good models, they are especially instruction optimizers and do not try to optimize directly the data or the data-cache. *ADORE* (ADaptive Object code REoptimization) [11] is a system that inserts prefetch instructions into the source code to achieve a speed-up. It is also a dynamic system that uses the same ideas than *Dynamo* since it also interprets the executed code and, when it notices that the execution is remaining in the same code segment, it will try to optimize it, store the new version in a shared memory block.

The major differences between *ADORE* and Dynamo are:

- *Dynamo* uses a certain number of counters (instrumenting the code) to know if a zone is being heavily used. *ADORE* uses the hardware counters that are in the *Itanium* processor to know the program counter position, the number of cache-misses in this zone, etc;
- *Dynamo* optimizes instructions, whereas the *ADORE* system handles memory issues. It inserts the prefetch instruction *lfetch* in loop nests, which will reduce the program's cache misses.
- *Dynamo* will try to optimize a lot of portions of code to achieve a speed-up. *ADORE*'s overhead being fairly low, it can concentrate it's efforts on the portions of code that can really be optimized.
- *Dynamo* can run in theory, on any kind of architecture. *ADORE* can only be executed and tested on the *Itanium* processor since it uses hardware counters and the *lfetch* isntruction.

Eventhough their system handles only loop nests, it is still a good automatic system. In the different programs they tested, they achieve a 20% speed-up while compiled at the third optimization level (compilation option -O3). The drawback being that it depends entirely on the *Itanium* infrastructure.

1.1.4 Another dynamic system under *Windows*

In 2002, Chilimbi and Hirzel [5] present a dynamic prefetching mechanism that uses two existing programs: *Vulcan* and *Sequitur* [13, 14]. It's with these two tools that they are capable of implementing a system capable of executing profitable prefetches.

Vulcan is a tool capable of modifying binary code even in the case of multi-threaded programes.

Sequitur lets us study the data accesses and extract patterns by constructing a grammar. It's by using both tools that they are capable of identifying interesting loads and inserting the prefetches.

Their system is not totally portable since the tool *Vulcan* can only work under the operating system *Windows*.

Chapter 2

ELF, Itanium, Modifying code dynamically

In this chapter, we will present the ELF Format and the Itanium processor. Of course, only the aspects in regard to the work done will be dealt with in detail.

2.1 ELF Format

The executable and linking format (ELF) was originally developed by Unix System Laboratories.

There are three main types of ELF files: executable, relocatable, and shared object files. These file types hold the code, data, and information about the program that the operating system and/or link editor need. The three types of files are summarized as follows:

* An executable file supplies information necessary for the operating system to create a process image suitable for executing the code and accessing the data contained within the file.

* A relocatable file describes how it should be linked with other object files to create an executable file or shared library.

* A shared object file contains information needed in both static and dynamic linking.

The ELF (Executable and Linking Format) Header is the only section in the binary code that has a fixed position. The other sections are indexed by the ELF Header. In a more general view, the ELF Header is a description section, giving the type of the object file (relocatable, executable, shared or core). But it is also an index table, giving the program's layout and the first execution instruction. All the other sections can be put in any order and may not even be present.

A more detailed view of the ELF Format can be found at [1].

Before continuing the presentation with the *Itanium* processor, we will just explain why the comprehension of the ELF Format is important for this work. We are trying to modify and optimize code dynamically, this means that we must know the basic structure of the binary code. Among other things, the ELF Format gives us the position of each function and their size.

2.2 Itanium

The Itanium processor [7] uses the instruction set called EPIC. The instructions are packed into bundles of three instructions. With certain restrictions on which instructions can be executed in the same bundle there is a possibility of executing the 3 instructions of the bundle at the same time. The execution units, the scheduler and the EPIC instruction set lets the Itanium execute up to 20 instructions in a single cycle.

127 … 87	86 … 46	45 … 5	4 … 0
instruction slot 2	instruction slot 1	instruction slot 0	template
41	41	41	5

Figure 2.1: Bundle format under the Itanium processor. Three instructions, each 41 bits long, and a 5 bit template

Instruction type	Description	Execution Unit Type
A	Integer ALU	I-unit or M-unit
I	Non-ALU Integer	I-unit
M	Memory	M-unit
F	Floating-point	F-unit
B	Branch	B-unit
L+X	Extended	I-Unit or B-unit

Table 2.1: Possible instruction types for the EPIC instruction set used in the Itanium processor

Of course, the Itanium-2 processor is a more complex processor. The instructions are still bundled together but there are more execution units, making it possible to execute more instructions in a same cycle. This section will present in more detail the itanium bundle, the general format of a bundle and of an instruction. Then we will explain the existence of templates for each bundle and the IP-relative branches used in jumps and calls.

2.2.1 Bundles

An instruction must be coded inside a bundle when using the Itanium processor. Each bundle is made of three instructions and, when no dependences are involved, the three instructions will be issued and executed at the same time. This whole section is a general view of the third part of *Intel Itanium Architecture Software Developer's Manual* (in no way do we suppose that this section covers the whole part of the manual, it is only a general presentation).

General format

The general format of a bundle is given by the figure 2.1. Each bundle is made of 16 bytes (thus 128 bits) and are divided in four parts. The first three parts are the three instructions (each formed of 41 bits) of the bundle and the last part is the template (last 5 bits).

It will be noted, from the programmer's point of view, the first byte contains the 5-bit template and 3 bits of the first instruction, the second byte contains the 8 next bits of the first instruction and so forth. A little bit of reflection is needed before tackling and trying to interpret a bundle using general C programmation.

2.2.2 An instruction code

Each instruction is categorized into one of six types. Table 2.1 shows the six different types. Each instruction is defined as 41 bits in the bundle. Figure 2.2 shows the basic instruction format. The basic instruction structure is:

1. The opcode is what defines exactly what is the operation that is going to be done. We define a very large definition of opcode since we include the possibility of hints, the possibility of a sign bit for the immediate (if there is one).

opcode	r3/f3/imm	r2/f2/imm	r1/f1/imm	qp
14	7	7	7	6

Figure 2.2: Instruction format under the Itanium processor. Generally the instructions use this format. There are, of course, exceptions.

Template	Slot 0	Slot 1	Slot 2
0	M−Unit	I−Unit	I−Unit
1	M−Unit	I−Unit	I−Unit
9	M−Unit	M−Unit	I−Unit
b	M−Unit	M−Unit	I−Unit
1d	M−Unit	F−Unit	B−Unit

Figure 2.3: A few templates to show the different possibilities. A double vertical lign represents a *stop*. The values of the template are in the hexadecimal base.

2. The next three portions are either register numbers (integer or floating-point) or a constant (also known as an *immediate*). There are almost every size of immediates with the general size being 8 or 22.

3. The last 6 bits is the *qp* portion. This is the predicate for this instruction. Predicates are used to define if this instruction is to be executed or not. If $qp = 0$ then the instruction is always executed, otherwise it depends whether the predicate is true or false. This is done using a *compare* instruction.

To know of which type the instruction is, the processor uses the template portion of the bundle and the opcode of the instruction. Once again, this is the general format, there are exceptions and the Itanium manual defines each instruction in detail.

Templates

The template lets the processor know which type of instructions are present in the bundle but it also gives us the *stops* that can be present in the bundle. The template is defined using 5 bits, thus defining 32 possible combinations of 3 instructions. There are actually only 24 combinations since 8 combinations are reserved. And, if we ignore the *stops*, we only have 9 different kinds of bundles.

We will now take the time to explain what is a *stop*. Since the itanium processor executes the instructions of each bundle in a parallel manner, it is possible that there is a data-dependence involved. The *stop* permits to put two instructions that depend on each other (one MUST be executed before the second one) in the same bundle.

Stops also have a more general use. Since the scheduler can schedule more than one bundle in every cycle, data-dependencies can exist between different bundles. Stops are also used to permit the execution of the first ones before executing the second one. Thus, eliminating any possible conflict.

Figure 2.3 shows a few templates that are possible.

IP-relative branches

One important detail that we will need to keep in mind is the way jumps and calls are designed in the Itanium processor. Jumps and calls are generally IP-relative jumps and calls; we will call *offset* the value of the jump.

This means that the destination calculation depends on the instruction address. One important thing to note is that the offset in the instruction is the number of *bundles* (and not bytes in the text segment) we must jump.

For example, if the jump has an offset of **3**, the program will skip 2 bundles and resume execution at the third bundle. Another example is using an offset of **0** which will define a infinite loop.

2.3 Modifying code dynamically

Now that we have presented how to find a function and how the instructions are coded on the *Itanium* processor, we will present a simple example of modifying a function. This was actually the first implementation showing that modifying dynamically code is possible. We could call it *Version 0*, the different low-level mechanisms that have to be put in place to make the whole system work should be self-explanatory.

We will not however show the source-code but will only show the general idea of the process.

2.3.1 The original example

Let us suppose that we have the function given by the table 2.2.

void ex()
{
int a;
a = 4;
printf("a: %d",a);
}
	alloc r34=ar.pfs,5,3,0
	addl r35=88,r1
	mov r33=b0
	mov r36=4
	ld8 r35=[r35]
	br.call.sptk.many b0=_init+0x170
	mov.i ar.pfs=r34
	mov b0=r33
	br.ret.sptk.many b0

Table 2.2: A simple example and its associated assembly code. For clarity, the bundles have been taken out, as have been the *stops* (represented by ;;).

What we will be doing is modifying this function so that the *a=4* becomes *a=6*. We will suppose that we know which function to modify (only the name).

Therefore, the Algorithm 1 gives the general view of what is done. We can see that it is with the ELF Format that we will be able to get the address of the function in memory. Then we have a known problem, it is of common knowledge that the text segment (the code portion of a process' memory) is a read-only segment. How can we then modify it?

- The text segment of a process' memory is allocated using the function **mmap**. The protection flag (read, write and execute) can be modified by the function

Original Bundle:	Assembler code:	New assembler:	New Bundle:
mov r36 = 4;; ld8 r35 = [r35] nop.i 0	0a 20 11 00 00 24 30 02 8c 30 20 00 00 00 04 00	0a 20 19 00 00 24 30 02 8c 30 20 00 00 00 04 00	mov r36 = 6;; ld8 r35 = [r35] nop.i 0

Figure 2.4: Binary instruction change. We show a *add* instruction in it's binary form (in the hexadecimal base), before and after the change. There is only one change between the two bundles and we have put in bold.

mprotect. The simplest solution is to *unprotect* the whole code-segment and then to *reprotect* it. Both functions are in the C library and have detailed man pages so we will not explain more than just giving their prototypes:

1. void * mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);
2. int mprotect(const void *addr, size_t len, int prot);

> Stop execution;
> Open binary file;
> Read the ELF Header and locate the *ex* function;
> Let ***fct*** be the address of the function *ex*;
> Let ***bund*** be the address of the bundle to change;
> Take off the write protection for the function ***fct***;
> Modify the bundle ***bund***, the instruction **mov r36=4** will become **mov r36=6**;
> Put back the write protection ***fct***;
> Resume execution;

Algorithm 1: General algorithm

Let us explain a little bit more in detail the various steps of the algorithm:

- *Stop the execution:* This can be done by either creating a separate thread that halts the execution of the main thread or, more simply, just adding a function call to the Dynamic Optimizer at the beginning of the main;
- *Read the ELF Header and locate **ex** function:* using the documentation on the ELF Header, this stage is just a traversal of a data structure. Finding the name of the function is just a question of parsing what is called the String Table;
- *Take off the write protection:* As it has been said, if we know the size of the executable file, we can use the function ***mprotect*** to change the write protection;
- *Modify the bundle:* This is just a question of finding the assembly instruction that we want to change. Figure 2.4 shows how the change is done.
- *Resume the execution:* Either the thread is terminated or the function returns.

Chapter 3

General overview

This chapter is a general overview of the project and of the purpose of the work done. We will explain the way the project has been started and the philosophy that was used to make the model more and more complicated and thus more general. Finally we will show two examples of programs that can be optimized with the current model.

3.1 From simple to more complex

When the project was started, it was not sure if it was feasible or even realistic. The solution for the implementation of such a project is to start with the simplest example, see if we can optimize it and move to a more complex example. Ideally, the second solution must englobe the first solution, trying to keep the speed-up unchanged.

Of course, more complex the solution, more chances are that the model will have a higher overhead, thus reducing the speed-up.

3.2 Presentation of the two examples

3.2.1 First example

The first model was capable of optimizing a simple loop nest that exists in some real-world applications. Figure 3.1 is the code source of the first example that was optimized using a Dynamic Optimization scheme.

We will briefly comment the choice of the example. First of all, the function has an instruction generating a lot of cache-misses. The load that is required to retrieve the data *t[k][j]* is going to practically always produce a cache-miss. This, of course, is logical if we suppose that the array is allocated using the C standard (in the order of rows).

```
long ex1(int **t, int m, int n, int o)
{
long res = 0;
int i,j,k;
for(i=0;i¡m;i++)
  for(j=0;j¡n;j++)
    for(k=0;k¡o;k++)
      res += t[k][j];
return res;
}
```

Figure 3.1: The first example: a nested-loop causing heavy cache-misses.

```
List ex2(long key)
{
List tmp = main_list;

while(tmp)
  {
  if(tmp-¿key == key)
    return tmp;
  tmp = tmp-¿next;
  }
return NULL;
}
```

Figure 3.2: The second example: a link-list traversal

Next, the values of m,n and o are not known at compile time and, since they can influence the choice of optimizing or not (will the initial overhead be covered by the ulterior speed-up?), only a dynamic solution can be used.

It is true however, that this is only a test example. An actual function coded this way has no real meaning, since a single traversal of the array, and in the right order, multiplied by m would be much quicker. Note, however, that the external loop only simulates successive calls to the function. A simple modification in the first model would be able to handle such a case. We have kept this version for reasons of clarity.

3.2.2 Second example

Probably as classic as the first example, a key search in a link-list structure is by far a good example to try to optimize. This example, that can be seen with figure 3.2, is however, more complicated because there are now two loads that we have to optimize: *tmp-¿key* and *tmp-¿next*. This example is already more complicated and it will be shown that the first model is not capable of taking care of it. However, the second model will be able to optimize it and, at the same time, handle the first example too.

Chapter 4

Version 1

In this chapter, we will present the first version of the Dynamic Optimization model. It is important to understand this model to be able to fully grasp the second one. This chapter (and the next one) are both written the same way. First, we will present some assumptions concerning the properties of the data access sequences. Next, we will present the model and how it works on a simple example. Finally, we will show a few implementation issues and a few results.

4.1 Assumptions for the data accesses

For the first model, that will only work on the first example, the assumptions are:

1. The data structure is a **read-only** data structure in all the scope of our dynamic code transformation.
2. The data sequence is always the same (there is no change in the order)
3. The data accesses are in a nested-loop, the outer loop determining when we restart the sequence

These two assumptions are very strict and, we will later show how we can try to make them more flexible. Once again, if the model is not able to optimize this example, there is almost no reason to try to complicate it, since the overhead will be even worse, thus giving us an even bigger speed-down.

4.2 Presentation of the model

All the assumptions simplify our model because it is then possible to monitor (and copy) the data in the first iteration of the external loop and then we will know exactly what will be accessed during the rest of the loop nest.

This is the building block of the first model.

To be able to model the data accessed, we then suppose we have a buffer large enough to copy all the data. Once again, these suppositions are made to simplify the model at an extreme to be able to determine whether it is worthwhile to continue.

Algorithm 2 shows the idea behind the first version. We can see that it is relatively easy and straightforward. The beginning is the same as in the first algorithm. We will not discuss in this report **how** we know which bundle is interesting or needs to be optimized (we could suppose a profiling scheme was used), this will be the main interest of future works.

Since we are going to unroll a loop nest, the resulting function size will be greater than originally. This means we cannot just leave the function in place (otherwise we would write on the next function that is in the binary file). Figure 4.1 shows the problem. The adopted solution is to copy the function to a *function buffer* and change the first bundle of the original function into an unconditionnal jump.

Stop execution;
Open binary file;
Read the ELF Header and locate the*ex* function;
Let ***fct*** be the address of the function *ex*;
Let ***bund*** be the address of the load that causes cache-misses;
Copy ***fct*** to the function buffer;
Modify the nested loop containing ***bund*** so that the first iteration of the external loop is extracted. The first iteration will be used to copy the data, the other iterations will use the copied data instead of the original data. Modify the IP-relative jumps;
Take off the write protection for the function ***fct***;
Change the first bundle of ***fct*** into an unconditional jump to the modified function;
Put back the write protection ***fct***;
Resume execution;

Algorithm 2: First version algorithm

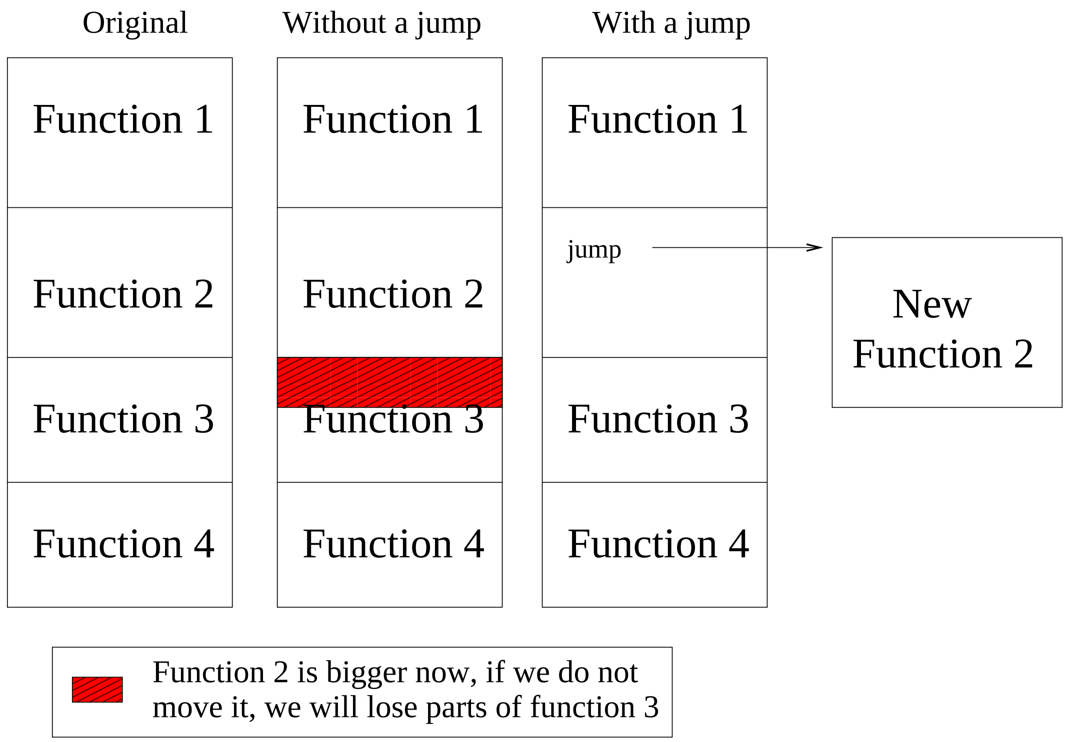

Figure 4.1: Function overflow. From left to right, original function layout in the binary file, if we add instructions to function 2, it will extend onto the third function (thus erasing parts of function 3), a good solution: using a jump.

This has a considerable advantage: if, for any reason, we would want to undo the optimization, a simple copy of the original bundle will reset the function.

Once we have copied the code of the function, we must now find a way to copy the *original* data. The most important idea to be able to this very quickly is to use the assembly instruction *st8 [r31] = r14,8*. We suppose **r14** is the register containing the data to store (the data that causes heavy cache-misses), and **r31** the register that contains the current address to the **data buffer**. The *8* at the end of the instruction instructs the processor to increment the value of **r31** by 8 after the execution of the instruction, thus positionning **r31** to the next free position in the buffer array. Using this instruction, we only have to initialize r31 to the beginning of the data-buffer. If that is done, this unique instruction is sufficient to copy the whole data sequence.

The last remark for this algorithm is the modification of the IP-relative jumps. There are two categories of jumps when moving the function in memory. Function calls also use IP-relative branches so any call to *printf* for example needs to be modified since we have moved the bundle containing the jump. The second category

<table>
<tr><th>Original code in C</th><th>Original code</th><th>First iteration</th><th>Other iterations</th></tr>
<tr><td>int ex(int *t,int m, int n)</td><td colspan="3">mov r8=r0;;</td></tr>
<tr><td>{</td><td colspan="3">mov r15=r0;;</td></tr>
<tr><td>int i,j,res=0;</td><td colspan="3">cmp4.lt p6,p7=r8,r33;;</td></tr>
<tr><td></td><td colspan="3">(p07) br.ret.dptk.many b0;;</td></tr>
<tr><td>for(j=0;j¡m;j++)</td><td colspan="3">sxt4 r14=r15;;</td></tr>
<tr><td></td><td colspan="3">shladd r14=r14,2,r32;;</td></tr>
<tr><td></td><td colspan="2">ld4 r14=[r14];;</td><td></td></tr>
<tr><td>for(i=0;i¡n;i++)</td><td></td><td>st4 [r28] = r14,4;;</td><td>ld4 r14 = [r31],4;;</td></tr>
<tr><td></td><td colspan="3">add r8=r8,r14</td></tr>
<tr><td>res += t[i];</td><td colspan="3">adds r15=1,r15;;</td></tr>
<tr><td></td><td colspan="3">cmp4.lt p6,p7=r15,r33</td></tr>
<tr><td>return res;</td><td colspan="3">(p06) br.cond.dptk.few 4000000000000630 ¡ex+0x20¿</td></tr>
<tr><td>}</td><td colspan="3">br.ret.sptk.many b0;;</td></tr>
</table>

Figure 4.2: A loop nest example. The bold instruction is the load that we have to optimize. We have taken out the *nop* instructions for clarity. If there is no vertical separtion, that means the instruction is the same for the different versions.

consists of jumps that jump over the modified section, the jumps must take into account the new distance to their destination.

The only real technique is to parse the function and check each jump, this will be one of the bottlenecks of the model, especially if the function that is treated is huge.

4.3 Example

To show exactly how the model works we are going to show how it reacts during a simple example. Figure 4.2 shows a simple portion of assembly code that sums the elements of an array. We use a predicate to test the result of the compare. Of course, there are multiple ways of programming a loop next in assembly code.

More and more compilers, and that includes the Itanium processor, have techniques to handle correctly and efficiently these kind of problems. The example we give is an understandable version given by the *gcc* compiler using the O1 optimization level. We have kept this one because of its relatively easy assembly code.

The assembly code is divided in three colums. On the left is the original assembly code, in the middle the first iteration of the external loop and, on the right, the assembly code for the next iterations. If we look more closely, we see that all we have to do to put this optimization in place is to:

1. Find the loop nest code that contains the data-cache miss instruction
2. Copy the beginning of the function before this loop-nest
3. Copy a first iteration of the loop-nest without the external loop jump
4. Add a *st4 [r28] = r14,4* instruction after the load
5. Copy the loop nest code for the other iterations
6. Copy the rest of the function
7. Add an unconditionnal jump to the new function

Original assembly code	New assembly code
	ld8 r14 = [r14] ;; st8 r31 = r14,8 nop.i
ld8 r14 = [r14];; add r14 = r14,r34 nop.i 0	add r14 = r14,r35 nop.m 0 nop.i 0

Figure 4.3: Instruction dependence in a loop. On the left the original bundle, on the right the load is extracted and put before with a store. The ; ; represents a stop.

4.4 Implementation issues

There are a few implementation issues that need to be resolved to have a fully working version.

4.4.1 Finding the loop nest code and extracting the first iteration

This is not always as easy as it sounds. As we have already stated in the previous section, depending on the compiler/processor, there are different ways to implement and program a loop nest in assembly code. For example, the *Itanium* processor has a system where the loop bundle counter is in a special register, that way, when there is a loop jump, the processor internally decrements the counter.

If this is the solution used, it is difficult to extract the loop nest (but not impossible). A solution would require to find the instruction that sets the counter and substract 1 *prior* to the loop nest execution.

Since we were only working on a prototype, the model is not able to do this. Furthermore, we want a more general model so this problem is actually irrelevant as we move to the second version. But it is interesting to note that extracting an iteration is not always as easy as it might seem.

4.4.2 Dependence in the bundle

If we look more closely to a bundle, as we have in figure 4.3 (on the left), we can see that adding a store after this load is not as easy as it might seem. There is a data dependence between the load and the add instructions. This means that if we simply add after this bundle another bundle containing the store instruction, it will store the data of the array, to which we have added something.

We have supposed that the data loaded from the array is always the same, not what we subsequently add to it. This means that we cannot just add the store instruction after the bundle. The contents of the register **r34** can differ between iterations.

What we need to do is extract the load and put it in a bundle before with the store, taking care of setting a stop between the two (in the figure on the right).

Of course, it is also possible for the load to be dependent of an instruction in the current bundle, as we can see in the figure 4.4. In this second example, the load does not need to be extracted because we can insert the store in the next bundle.

These two simple examples show that it is not always easy to know *how* to insert the store instruction. A special care must be taken to make sure that we do not change the semantic of the program.

4.5 Results

The results of this model on the example in figure 3.1 was surprisingly good. We knew before hand that any model that tries to relocate data and optimize data-cache

Original assembly code	New assembly code
add r14 = 24,r15;; ld8 r14 = [r14] nop.i 0;;	add r14 = 24,r15;; ld8 r14 = [r14] nop.i 0;;
	st8 r31 = r14,8;; nop.i nop.i

Figure 4.4: Instruction dependence in a loop, version 2. This time, the store can be easily insterted in the next bundle of the code.

miss *had to* get good results on such an unfriendly case for the cache-hit problem. Our tests showed that for almost any values of m,n and o we got more that a 50% speed-up. In some cases, we got up to 78%.

Eventhough we could make a table with the results, we think it is useless since any value of the parameters give the same results. This is true regardless of the size of the array used. However, we used the word *almost* since it is impossible to test every possible values of m, n and o.

What has to be remembered from this first version is that it is a very simple model which is able to find loops and optimize a load by copying the data accessed during the initial loop and then use the buffer cache instead of the initial data. This first prototype was implemented to prove that such a technique can with the speed-up that was obtained, we believed that we could complicate the model to be able to cover more complicated cases. This lead us to the version 2 of the model.

The proof of correctness of the model can easily be obtained with the assumptions that were enumerated. Indeed, we admit that our assumptions in section 4.1 on page 12 are very restrictive. The data is a **read-only** data structure, this means that once copied, we know that there will be no change, hence no updates will be needed. The fact that the data sequence is always the same means that by knowing where is the start, we can easily reset the model so that it can load the first data in the sequence. And, finally, the load is in a loop nest, this means we **know** where the data accesses are going to cycle.

Chapter 5

Version 2

The second version of the model was created in such a way that it is able to optimize the first example and can handle examples the first model cannot. It can be defined as being a more general model that the first version but, at the same time, maintaining a good speed-up on the first example.

This chapter will have the same organization as the previous one. We will, however, take the time to show and explain the differences between the two models.

5.1 Assumptions for the data accesses

This second version is more complicated than the first model. This also means that it is capable of handling more complex examples. This model will also be able to handle the first version and almost as well.

These changes in the model reduce the assumptions that were put up for the first version, these are the new assumptions:

1. The data structure is a **read-only** data structure.

2. The data sequence has a certain redundancy that will be explained later.

As we can see, the assumptions are simplified. The *read-only* is kept since it reduces the number of checks of data-consistency to a minimum. The second assumption has to be detailed. Let us try to explain what we mean by *a certain redundancy.*

This second model is still relatively restrictive on the kind of data accesses that can be modeled. After being able to handle array accesses in a loop-nest code, we wanted to tackle the link-list problem. The second example, given in figure 3.2, show a basic search for an element in a link-list depending of the key.

This time, there is no proof or hint telling us when we are going to be restarting the traversal of the data structure. We will use the address of the first element as a reference to know when we are restarting the traversal. We suppose the link-list is a simple link-list with no cycles. The fact that we say that the data structure is *read-only* means that nothing in the link-list changes (not the key values, the element values or the links).

5.2 Presentation of the model

Knowing that the data-structure is *read-only* helps us to create a good model for this problem. We will use a finite automaton to define the model and know when to restart the data sequence. Figure 5.1 shows the model as a simple state machine. The initialization phase is to have the first address as a reference.

The model will optimize the load instruction. We will leave the calculation of the address intact, that way we can compare the current address with the first address.

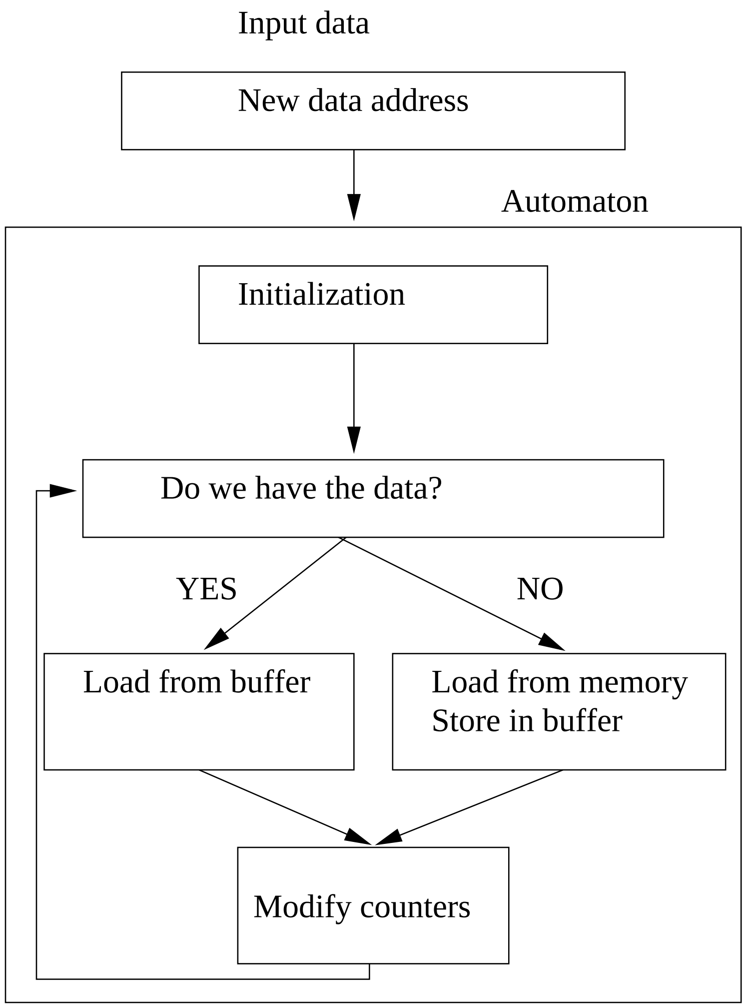

Figure 5.1: Finite state machine for the second model. The box on the top is the data that is fed to the state machine.

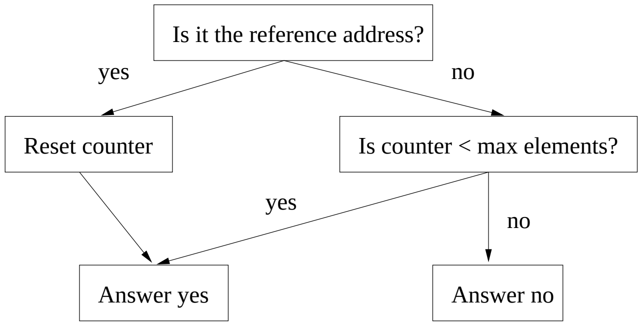

Figure 5.2: Finite automaton to answer the question "Do we have the data?"

The two big questions that we will have to consider to are *Do we have the data?* but even more important is *Where is that data?*. These two questions will remain the corner stone of the next versions that will be implemented. Not only must we be able to answer them but we have to do it as fast as possible, since we are working at the same time as the program. We do not have the luxury of time that static optimizers have.

5.2.1 The *Do we have the data?* dilemna

The figure 5.1 has a "black box" in the middle. The question *Do we have the data?* is where all the problem resides. If we can answer that question efficiently and give the data requested without a cache-miss, we will obtain a speed-up. In our case, we use 2 counters and 2 pointers to answer the problem.

Luckily, with the assumptions for this model, the two questions are quite easy to answer. *Do we have the data?* can easily be answered using two counter and an address reference. Figure 5.2 shows the automaton that gives us the answer to that question. As we have already said, we suppose the initialization phase has set up the reference address and put the counters to zero.

Then, once we have the answer to the question, we must take action accordingly (Let us notice that if the address is the reference address, we only have to load the data and reset the counter):

- If we do have it, then we load the data from the buffer, increment the counter.
- If we don't, it means that we must add it to the buffer. We store the data into the buffer, increment the counter and the max.

5.2.2 The *Where is the data?* question

Once we have answered the question of whether or not the data resides in our own data-buffer, it is important to figure out how to load it. More specifically, we need to know exactly where it is. This is where our assumptions help us. Their restrictive nature gives us an easy solution to this problem.

Since we know that we have a certain redundancy in the data sequence, we will know straight away where the data resides. Even better, with almost no work, we will almost always have a register at the right place, no extra calculations will be needed.

Let us suppose that the register r31 is a pointer referencing the data-buffer. At the initialization step, the model will store the first data and shift the register r31 so that it points to the second element of the data-buffer. These are the different possibilities for the register r31:

1. If the next access is the first element, we only have to put r31 back at the beginning of the data-buffer (only one *mov* is needed). Then we can load the

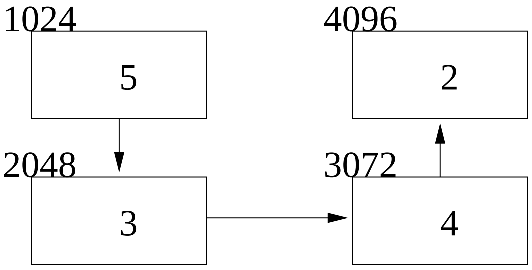

Figure 5.3: A simple link list example, the information around the cells are their addresses, the information in the cells is the key value.

ex2(5);	1024
ex2(3);	1024 2048
ex2(5);	1024
ex2(4);	1024 2048 3072
ex2(2);	1024 2048 3072 4096

Figure 5.4: The sequence of calls to the second function and the addresses of the data structure accessed

first element using the instruction ”ld8 r15=[r31],8”. This instruction will load the first element and then move r31 to the second element of the array for an possible store in the future.

2. Otherwise, we have a new data that needs to be stored. We will want to put it next to the first one and r31 is exactly there! Therefore, only a store ”st8 [r31]=r14,8” is needed.

This means that the value in r31 is always up-to-date and is ready to either store (if we are at the end of the data-buffer) or to load (if we are in the middle of the data-buffer). The use of ”st8 [r31]=r14,8” or ”ld8 r15=[r31],8” gives us, in one instruction, the load or store and the incrementation of the value in register 31.

5.3 Example

As an example for this model we will give a simple sequence and the state of the model at each step. The example will be a sequence of data accesses that traverse a link-list given by figure 5.3. This is a very simple link-list. We also suppose that the function given by figure 3.2 is going to be called more than once. Figure 5.4 gives the sequences of calls made and the different addresses that were accessed. For this example, we will only handle the *tmp-¿key* instruction.

5.3.1 Initialization phase

When the first access will be needed, the model will set up the pointers and counters as it is needed for the rest of the execution. This is called the *Initialization* phase.

Sequence:	**1024** 1024 2048 1024 1024 2048 3072 1024 2048 3072 4096
State:	1024 1 1
Data-buffer:	5

Figure 5.5: The sequence of address accessed

In our example, figure 5.5 gives the data access order as it is defined by the figure 5.4 and the state of the model . Throughout the example, we will give different figures with the sequence, we will also put the current address in bold. The state of the model is given by 3 integers, the first one is the reference address that will not change throughout the example. The second integer is the counter and the third one is the number of elements in the data-buffer.

As we can see on the third line, we have a single 5, which is the first key of the link-list. This is the first element of the data-buffer. When we will need the first element of the link-list (for the key value), we will be able to use this integer instead of loading part of the node from memory.

5.3.2 Next 3 accesses

We will show in detail what happens next.

Sequence:	1024 **1024** 2048 1024 1024 2048 3072 1024 2048 3072 4096
State:	1024 1 1
Data-buffer:	5

Figure 5.6: The sequence of address accessed (2)

Figure 5.6 shows the state after we have treated the next address. As we can see, the model has not moved. But something has been done, the question *Do we have the data?* actually answered yes. Since the next address is also 1024, we know we have the data. So we reset the counters and when we load the first element (here 5), we increment the counter back to 1 (for obvious reasons, we do not really put the counter to 0, because we know that in the end, it must be equal to 1... We directly assign 1 to it's value). Of course, when we reset the counter, we also reset the register r31 that points to the current cell in the data-buffer.

Sequence:	1024 1024 **2048** 1024 1024 2048 3072 1024 2048 3072 4096
State:	1024 2 2
Data-buffer:	5 3

Figure 5.7: The sequence of address accessed (3)

Figure 5.7 shows the evolution of the model when we add something to the data-buffer. We knew we did not have 2004 in the buffer, so we had to store it there. That is why the counter and the max are now at 2 and that there is a 3 in the data-buffer.

Sequence:	1024 1024 2048 **1024** 1024 2048 3072 1024 2048 3072 4096
State:	1024 1 2
Data-buffer:	5 3

Figure 5.8: The sequence of address accessed (4)

Figure 5.8 shows us what happens when we encounter a 1024 (which is the reference address). As we can see, the only difference is the counter that is reset one more time. Nothing else really changes.

5.3.3 And finally...

If we walk through the sequence adding to the buffer, reseting the counter when needed, the last access will leave the model in this state: We notice that we now

Sequence:	1024 1024 2048 1024 1024 2048 3072 1024 2048 3072 4096
State:	1024 4 4
Data-buffer:	5 3 4 2

Figure 5.9: The sequence of address accessed (Final)

have every key of the link-list and, to now do the search, we only have to look at 4 integers instead of walking through a whole link list.

5.4 Implementation issues

There were a few implementation issues when we first tried to put up this model. We will present them briefly here.

5.4.1 The instruction cache

When working dynamically and especially when modifying code at run-time, the instruction cache has an important influence.

When we modify the binary code, the problem is that the change might not be taken into account straight away if the processor is working with the cached version.

This means that the changes must happen before hand or we must accept to wait until they will be. We tried to find ways to flush the instruction cache but to no avail.

Another important note is that this reduces the options we have in modifying the code. If our modification is only on one bundle then there is no problem. But if we want to change, remove more than one bundle, the program will probably crash. Since, the cache is not aware of the changes, it can ask to reload part of the modifications and not the totality. This means that for a certain while, the program can be running part of the original code and part of the modification!

The only reasonable solution is to overwrite a single bundle with a jump if the modifications are on more than one bundle. There are always synchronization problems if we are working with threads. The write is only atomic for a 8-byte write so that means that the thread could be paused while changing a bundle. In the same manner, if working with threads, the only solution is to pause the main thread before overwriting the bundle.

At first, we wanted to include the initialization code and, when it was executed, overwrite it with the main stream code. This would have reduced the number of required tests. But, because of the instruction cache, this was impossible. If a solution is found, this optimized solution would be much better.

5.4.2 Scratch registers

At first, the idea behind this model was to use 4 global registers (r28-r31). But the problem with this solution is that, even if they are called *global* registers. Between function calls, they can be reset and put back to zero. This means that it is not possible to use them in programs where there are function calls. The model must save their values into local registers if possible or spill the values into memory.

Another big problem is when the model needs to optimize more than one load. In this case (as it is in the second example), then again, local registers (but how many can we use?) or spilling is necessary. This means adding more bundles and thus rising the overhead.

The problem of using excessive local registers or spilling the data is not to be taken lightly. The success of the model depends on its light-weight characteristic.

5.5 Results

Comparing with the results of the previous model, this model is almost as good. Of course there is a little bit more overhead but it does obtain the 50% speed-up. What is more important is that it also optimizes this example.

There hasn't been a so good speed-up with the link-list example, but the model must optimize two loads instead of only one and there is more overhead. The results still show that this solution is a good one. It needs to be broadenned to handle even more general examples but it is a good start. This model is capable of handling any load as long as they respect the redundancy assumption, but it can be located in a loop, in a recursive function or just a regular function. That aspect of the model is already a big step compared to the first version.

Chapter 6

The next step

In this final chapter, we will try to explain why the work must go on and especially in what direction. A lot of things have been done, a whole API has been put up to be able to locate a function, locate a load and modify code dynamically. This API will be used in the next versions and other works that will need to modify, interpret or verify binary code while a program is running.

6.1 Discussion on the remaining assumptions

The two remaining assumptions are:

1. The data structure is a **read-only** data structure.
2. The data sequence is has a certain redundancy that will be explained later.

The first assumption is put there to keep the model light-weight in the sense that, if it was not a read-only data structure, complicated tests and interpretations would be needed to verify and prove the correctness of the model. That assumption will likely be the last one to be removed.

Without the second one, the model will also become more complicated. It is thought that instead of removing it, we will try to render it even more flexible by redefining what we mean by *a certain redundancy.*. We are already trying to see if another solution, resembling a hash function, can not be used to locate quickly if the data is in the data-buffer or not.

6.2 Implementation issues

Of course, more complicated the model more assembly code will be required. This poses an important problem that will be exposed in two parts:

- When programming in assembler, we are almost at the lowest level possible (leaving binary programming to the secluded elite). The problem that arises is the lack of possible debugging. In fact, almost no debugging is possible, because we are programming dynamically and, for example, **gdb** does not react well if you try to debug a dynamically modified program;
- When the algorithm becomes more complicated, it might seem natural to just add a function call instead of the optimizer's code. But the problem comes from the famous scratch registers. Since we do not really know which ones are being used, we would have to store every scratch register (roughly twenty) before the call and load them back after the call. This will likely add a huge overhead that will make the whole optimizer cause a slow-down...

6.3 Where to from now and what about you?

The work that will be done in the immediate future is to try to study how can we recalibrate the assumptions to make the model more general. After that, we will be able to attack general benchmarks to test our model(s).

Of course, since all this has been a succession of prototypes and tests, a lot of optimizations are probably possible. It would be interesting, at a certain point, to launch a second team that would try to optimize the optimizer!

Eventhough we have not really shown pages of listings of our versions, we think that a general programmer can probably put the pieces together. If there are any questions, theoretical or on the implementation aspect of the problem, do not hesitate to send a mail!

Conclusion

We have shown exactly how a dynamic optimizer can be set up. We have tried to explain what kind of applications can be done and exactly how they would be programmed. The basic problem of dynamic programming is probably going against what the processor and hardware engineers are used to. Efforts to make a basic program execute in the fastest time possible have created things like the Instruction-cache, something that, without a possibility of a flush, will always be a problem for our work and our implementations.

The two examples on which we have built our models are somewhat simple but very representative. They can easily be found in any real application and it is for that reason that they were chosen. The basic motive of this work is to get a working general dynamic optimizer. Whether the final version will even remotely resemble the first or second version is not relevant, since to learn about dynamic optimization, it was probably necessary and a good exercise to put those two on their feet.

What is certain is that dynamic optimization is a solution to many statically untractable problems. There are things that will only be known at run-time and a dynamic process is sometimes the only solution. This work tries to show how modifying code is easy and light weight. The question of what do we modify and how, is where lies the problem and the big question mark.

But once we have the tools to change, extract, delete or add code during run-time, dynamic optimizations become a reality.

Bibliography

[1] Elf format. http://www.cs.ucdavis.edu/ haungs/paper/node10.html.

[2] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 280–290, New York, NY, USA, 1998. ACM Press.

[3] M. Annavaram, J. M. Patel, and E. S. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings of the 28th annual international symposium on Computer architecture*, pages 52–61. ACM Press, 2001.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[5] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 199–209. ACM Press, 2002.

[6] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. Deli: a new run-time control point. In *35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 257–268, December 2002.

[7] Intel(R). *Itanium(R) 2 Processor Reference Manual for Software Development and Optimization*, chapter Optimal use of lfetch, pages 71–72.

[8] D. Kim, S. Liao, P. Wang, J. del Cuvillo, X. Tian, X. Zou, D. Yeung, M. Girkar, and J. Shen. Physical experimentation with prefetching helper threads on intel's hyper-threaded processors. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization*, pages 27–38, 2004.

[9] J. Kim, K. V. Palem, and W.-F. Wong. A framework for data prefetching using off-line training of markovian predictors. In *20th International Conference on Computer Design (ICCD 2002)*, pages 340–347, 2002.

[10] A. Krall. Efficient JavaVM just-in-time compilation. In J.-L. Gaudiot, editor, *International Conference on Parallel Architectures and Compilation Techniques*, pages 205–212, Paris, 1998. North-Holland.

[11] J. Lu, H. Chen, R. Fu, W. Hsu, B. Othmer, P. Yew, and D. Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 180–190, December 2003.

[12] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *28th annual international symposium on Computer architecture*, pages 40–51, 2001.

[13] C. G. Nevill-Manning. *Inferring Sequential Structure*. PhD thesis, University of Waikato, Nouvelle Zlande, 1996.

[14] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary Transformation in a Distributed Environment. Technical Report MSR-TR-2001-50, 2001.