Internship report:
Automatic data movement for performance enhancement on Cyclops64

Bertrand Putigny

From April 1st to October 1st 2010
Contents

Abstract 2

1 Institute 3

2 Internship overview 4
  2.1 Subject ................................................................. 4
  2.2 State of the art ......................................................... 4

3 Project 6
  3.1 Cyclops64 architecture .............................................. 6
     3.1.1 Overview ............................................................. 6
     3.1.2 Cyclops64 chip ...................................................... 6
     3.1.3 Simulators ............................................................ 7
     3.1.4 Cyclops64 programming and memory model ....................... 8
  3.2 Dynamic binary rewriting framework ................................. 9
     3.2.1 Motivations .......................................................... 9
     3.2.2 Overview ............................................................ 10
     3.2.3 Features .............................................................. 12
  3.3 Implementations ......................................................... 17
     3.3.1 Software cache ..................................................... 17
     3.3.2 Caching system on registers: ..................................... 20
     3.3.3 Prefetching .......................................................... 21
  3.4 Concurrent execution issues ........................................... 23
  3.5 Performances ............................................................ 24

4 Achievements and future work ........................................... 26
  4.1 Achievements ............................................................ 26
  4.2 Future work ............................................................. 26

Conclusion ................................................................. 27
  4.3 Personal benefits ....................................................... 28
  4.4 Acknowledgment ......................................................... 28
Abstract

I realized my 6 months internship of last year of engineering school in a research laboratory of the University of Delaware (UDel) in the United States. I was working in the Computer Architecture and Parallel Systems Laboratory (CAPSL). This main research of this lab is about high-performance parallel computing architecture.

The CAPSL laboratory is directed by the Distinguished Professor Guang Gao. And my adviser was Joseph Manzano, his senior PhD student. I have also been managed by J.C Beyler, doctor in computer science, working in ET.International, a computer technology and software company founded by Prof. Gao.

One of the main architecture the CAPSL laboratory is working on is an IBM multi-processor chip, Cyclops64, formerly Blue Gene/C. Therefore my work during this internship is based on this architecture.

Cyclops64 is a very good example of the trends of now-days supercomputers, with a great availability of computation power. Therefore, most of the applications have shifted from being computation bound to memory bound. Availability of massive computation power makes data layouts and memory accesses major factors in achieving performance for the new generation of multi or many-core chips. IBM’s Cyclops64 architecture exhibits a complex software managed memory hierarchy without any hardware data cache. This allows exciting opportunities for the programmer to exploit the memory hierarchy. However, for applications with high locality, this can prove more than a burden than a benefit. This is why a dynamic software cache can be beneficial for these types of applications.

This report presents my experiences working on an automatic data movement for the Cyclops64 architecture which uses dynamic rewriting to optimize the code. It present an extend-able framework which uses two common methodologies to speed up data access; a software cache based one and a prefetching based one. This provides two great advantages over similar optimization frameworks:

- Allows the framework to be fully transparent to the programmer and to be independent of the tool-chain, and
- Has the ability to learn and instrument (or de-instrument) themselves according to certain conditions on the given machine.

Although the software cache approach has many problems to overcome, i.e scheduling and line conflicts, it shows little to no performance degradation on the tested applications. For the prefetching method, tested applications show a maximum speedup of 35%.
Chapter 1

Institute

I did my internship in the University of Delaware (UDel). I worked in the Electrical and Computer Engineering department of UDel.

I was working in the Computer Architecture and Parallel Systems Laboratory (CAPSL) of the University of Delaware. This laboratory is managed by Professor Guang R. Gao, Distinguished Professor in the Electrical and Computer Engineering department of UDel.

This laboratory’s main research topics are about high-performance parallel computing architecture, system software, parallel programming and tools for supercomputers as well as high-performance embedded systems. CAPSL has a lot of connections with ET International, a computer technology and software company founded by Prof. Gao.

ET International is an independent software vendor, it has been founded in 2000 by Professor Guang R. Gao, Distinguished Professor of Electrical and Computer Engineering at the University of Delaware. There main activities are developing system software and tools on high performance architectures, especially massively parallel architecture. For instance, they are particularly involved in development of tools and software for Cyclops64, an IBM massively multi-processor chip.

I had two main supervisors, Joseph Manzano who is Dr. Gao’s senior PhD student and Jean Christoph Beyler working for ET International who did a very similar work but for an other architecture during his PhD.
Chapter 2

Internship overview

2.1 Subject

The initial subject of the internship was to design and implement a software managed cache for homogeneous multi-core architecture. That mean implementing a cache system on an architecture that doesn’t posses a hardware cache. The target architecture is IBM’s Cyclops64.

Because we do not have access to the source code of the compiler for Cyclops64, we need to use a binary rewriting technique. Therefore we had the choice to design a static binary rewriting framework, i.e implementing a tool that will read the binary right after compilation and create new binary with the caching ability. Or to design a dynamic binary rewriting framework that will rewrite the binary after the loader put it in memory to be executed.

We choose the dynamic solution because, it allow us to use different code versions, i.e we can have several versions of the code in memory and switch between them during the code execution. Moreover at run-time we have access to more information than for the static solution.

The main difference between hardware and software cache is that software cache has an overhead while hardware doesn’t (we have to execute more instruction to handle the software cache, while with a hardware cache, we still have the same number of instruction). Thus it is important to use the software only for code portion where it worth it. And that is why the dynamic solution so much better than the static one for us, because the information available at run-time will allow us to learn when it is worth to use the cache or not and the code versioning will even allow us to switch from using cache or not depending on run-time conditions.

2.2 State of the art

This part presents different works related to this contribution. Binary rewriting as been heavily used for numerous years, here are some applications using binary rewriting:
Pin by Intel:
It the closest work related to our contribution, Pin uses binary dynamic rewriting to instrument code. It provides a very easy way to write binary instrumenting tools for Intel architecture.

But Pin is just available for Intel architecture, not for Cyclops64, moreover Pin just allows you to write your own tools write a dynamic binary instrumenter. But it doesn’t provide any performance improvement tools. It mainly aim to help the design of profiling tools.

The other difference is the way it is instrumenting code. Pin instrument code by adding at each point chosen by the designer a call to a function written in C/C++ by the tool designer. This present a much bigger overhead than our implementation.

Our framework also provides code pattern analysis, instruction dependency detection and learning model not provided by Pin.

Valgrind:
Valgrind is available for the following platform: X86/Linux, AMD64/Linux, PPC32/Linux, PPC64/Linux, and X86/Darwin (Mac OS X). It is also a dynamic binary instrumenting framework. It is can be used for code debugging and code profiling, it is not designed to increase performances of instrumented code.

Esodyp:
This the work of Jean Christophe Beyler during his PhD. This work is also very similar to the one presented in this report, as it has a learning ability and is used for performance improvement, however it take advantage for hardware prefetcher to improve software performance. While our work is a complete software solution, even for architectures without hardware prefetcher.
Chapter 3

Project

3.1 Cyclops64 architecture

3.1.1 Overview

The Cyclops64 (C64) is an ambitious petaflop supercomputer project under development at IBM research Laboratory. C64 is designed to run high performance applications. It is attached, through a number of Gigabit Ethernet links to, a host system. The host system provides a familiar computing environment, through an operating system such as Linux, to software developers and users.

A Cyclops64 supercomputer is built out of tens thousands C64 processing nodes. A processing node is made of one C64 chip (its architecture is detailed in the next section), an external DRAM memory and a small amount of external interface logic.

Moreover, access to a file server is given to each C64 chip through the Gigabit Ethernet links. This file server can be used to store input and output data needed by each application. Beside this file system, each C64 chip can be connected to a serial ATA disk drive.

Each nodes are connected between themselves through a 3D mesh network. Figure 3.1 presents the overall architecture of a Cyclops64.

3.1.2 Cyclops64 chip

During the work presented in the report, we just focused on a single C64 chip. The figure 3.2 represents the C64 node architecture.

A C64 chip is composed of 80 processors, each of the consisting two thread units, a floating-point unit, and two SRAM memory banks of 32KB each. Hence, the total on-chip memory is approximately 5MB.

A 32KB instruction cache, shown in the figure 3.3, is shared among five processors. At boot time, SRAM banks are partitioned into two segments. One segment contributes to the globally shared interleaved on-chip memory. Processors and interleaved memory are logically arranged in a dance hall configuration with processors and memory banks on opposite sides connected by a one-level crossbar switch. The other segment, called scratchpad memory (SPM), constitutes the local memory. The corresponding thread unit has fast access to its own SPM. The C64 architecture also
provides four DRAM controllers. Each one is attached to a 256MB bank, hence a C64 node provides 1GB of off-chip DRAM.

A processor can access its own scratchpad memory in just 5 cycles, on-chip SRAM memory in at least 33 cycles and off-chip DRAM memory in at least 59 cycles. A thread unit is a simple 64-bit in order RISC processor core operating at 500MHz. Hence one single C64 chip can achieve 80 Gflops peak performance.

The figure 3.3 shows how a C64 supercomputer can be arranged to achieve more than 1 petaflop peak performance.

### 3.1.3 Simulators

Because in the laboratory I was working in we do not have access to a C64 chip, we are using different simulators instead of the real chip to test our software and different contributions.
We used two different simulators, the first one is FAST [2] (Functionally Accurate Simulator Toolset for the Cyclops-64). And we also used a more accurate emulator: DIMES: An Iterative Emulation Platform for Multiprocessor-System-on-Chip Designs [1].

3.1.4 Cyclops64 programing and memory model

Cyclops64 supports user and supervisor operation modes. Moreover, execution is non preemptive \textit{i.e} there is not mechanism to interrupt a running program once it has started. Cyclops64 is using TiNy Threads (TNT) [3] as a virtual machine for thread execution. Each software threads is directly mapped to a physical thread, so after a software thread is assigned to a hardware thread unit (TU), it will run until completion. Moreover a software thread will never be swapped out to allow idle hardware resources to be assigned to another software thread.

Cyclops64 provides a rich set of hardware support for synchronization through many memory atomic instructions. Moreover TNT provides classic synchronizations methods (like barrier, mutex...).

As explain before, Cyclops64 has three-level memory hierarchy, with no virtual memory manager, thus the memory hierarchy is fully exposed to the programmer. Cyclops64 architecture follows a sequential consistency model for the interleaved and off chip memories.
3.2 Dynamic binary rewriting framework

3.2.1 Motivations

Thanks to the massive availability of computation power, most applications have shifted from being computation bound to memory bound. Availability of massive computation power makes data layouts and memory accesses major factors in achieving performance for the new generation of multi or many-core chips. IBM’s Cyclops64 is a prime example of these next-generation chips. This architecture exhibits a complex software managed memory hierarchy without any hardware data cache. This allows exciting opportunities for the programmer to exploit the memory hierarchy. However, for applications with high locality, this can prove more than a burden than a benefit. This is why a dynamic software cache can be beneficial for these types of applications.

This report presents my experiences working on an automatic data movement for the Cyclops64 architecture, using a dynamic rewriting to optimize the code.

We present extendable framework which uses two common methodologies to speed up data access:

1. a software cache based one and
2. a prefetching based one.
The dynamic binary rewriting part of this project provides several great advantages over similar optimization frameworks:

1. Allows the framework to be fully transparent to the programmer,
2. Allow it to independent of the tool-chain, and
3. Has the ability to learn and instrument (or de-instrument) themselves according to certain conditions on the given machine.

Although the software cache approach has many problems to overcome, *i.e.* scheduling and line conflicts, it shows little to no performance degradation on the tested applications. For the prefetching method, tested applications show a maximum speedup of 35%.

### 3.2.2 Overview

The dynamic binary framework is a library which allow a running program to self optimize. The main reason we decided to use dynamic binary rewriting is because we don’t have access to the source code of the C compiler on cyclops64. Thus we cannot provide a modified compiler that will take care of data movement for programmers interested in this feature.

But using binary rewriting also provide several very interesting advantage. First of all it allow us to collect run-time information that cannot be detected by a static analyze. It also allow us to use code versionning and to switch from one version to another dynamically depending on different strategies.

It is important to understand that all actions done by this framework are done after that the loader put the program into memory. Thus the first action done by the framework is to parse the binary code of the running program. To do so, we need a step before executing the program: we need to disassemble the binary using objdump. We have a script that will create a file (name `fct_table`) in the working directory. This file contains the list off all functions of the program and there address. Thus by reading this file (at run-time) we have access to the address of each function in memory so we can parse each function one after an other.

The code generated by the framework is located in DRAM, like the program’s code but in different location, we do not insert the generated code in the middle of the program’s code. We do not insert code in the original code because inserting instructions inside the original code would change the address of the next instruction, and that would result in breaking all branch (because they would branch to other instructions).

While parsing the code we can choose to instrument some instructions. To instrument an instruction, we replace this instruction by a branch to some piece of code generated previously to handle this instruction. During code parsing we keep an instruction table that contains information on each
instruction we want to instrumenting.

The framework provides several functions to allow an “easy” code parsing and code generation. For instance, for code parsing, we provide functions to get the registers used by the instruction pointed, the instruction itself (if it is a load, a store,
a branch etc.). We also provide functions to generate binary instructions.

We can see on Figure 3.4 on page 11 how the instrumentation take place. As we can see, in a first phase before the user's program starts, we parse the code, generate our code and instrument the selected instructions. This description of the framework behavior is not completely accurate, it is the description of an older version, but is it also the easiest to understand. In a further section we will describe how it is done now, but the main idea is still the same.

3.2.3 Features
This section of the report aims to present one by one the different features of the dynamic binary rewriting framework.

Code parsing
When parsing the code, the framework has a complete view of the code, thus we have a lot of very interesting opportunities to get important information about the code.

During a first parsing phase we look for unused registers in the function being parsed. We keep in a function table the registers unused by the function. As explained in section 3.1, Cyclops64 has 64 registers but some of them are volatile, i.e., they are not preserved across calls. This are the unused registers we are looking for. Because if they are not used by the function itself, and as the others function do not expect a certain value from them we can use them freely, without the need to save and restore them.

The framework will also detect loops. We need several parse to detect loops. We detect innermost loops, so for each loops detected we need to parse the loop's code again to check if there is another loop inside the loops body. The loops detection works as follow:

1. Detect branch targets. During this parse we also detect unused registers.
2. Detect backward branches to one of the target found in parse 1.
3. For loops found in parse 2. check if there is no inner loop.

After detecting a loops, the framework will parse the loop's body to find instructions to be instrumented. As we focus on improving memory performance, in the current implementation we choose to instrument memory instruction i.e loads and stores. During this pass, choosing to instrument an instruction comes down to add it into the table that contains all the instrumented instructions.

When an instruction is selected for instrumentation, the framework will detect dependencies between the this instruction and the next instruction in the loop. To detect dependencies, the framework check if the next instructions use the register target of the selected instruction. The dependency check will stop as soon as a dependency is found. For instance if we have the code pattern shown on Listing 3.1
the framework will not find any dependence between instructions (1) and (2), but as it find a dependence between instruction (1) and 3, it will not check dependence between instruction (1) and (4). Thus it will just find that instruction (1) has no dependence with (2) even if it also has no dependence with instruction (4).

### Listing 3.1: Example of instruction dependence detection

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LOAD r8, 0(r9)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ADDI r9, r9, 4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>STORE r8, 0(r10)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ADDI r11, r11, 1</td>
<td></td>
</tr>
</tbody>
</table>

The framework will also try to insert a prologue to each function. That mean that it will search for a spot where to insert a jump right after the beginning of the function. In one case the insertion will fail: if the first instruction of a function of a branch with an offset. Indeed a branch with an offset depend on the address of the instruction itself. Therefore we cannot replace this instruction with a branch to our prologue and perform this instruction in our prologue because it would result in branching to an completely unexpected code location.

The prologue insertion is always required, for same implementation we do not need to add a prologue for each function. In section 3.3 on page 17 we will see why for in some case we need to add a prologue and why sometime we don’t need it.

We have chosen to just instrument the user’s code and instrument neither the library’s code nor the kernel or system code. We do not instrument the library code because it is not in this code that the application will spend most of the time so it is not worth to pay the overhead of instrumenting it for almost no performance gain. And we do not instrument the system’s code because this code should already be optimized, and therefore should not benefit the optimizations our framework can provide.

### Code generation

Once the code has been parsed and that the framework has selected the instructions to instrument, the code generation phase will take place. The code generation comes down to browsing the instruction table and generate the code accordingly to the kind of instruction. For each instruction on the table we will generate at least one piece of code.

The code will be generated in one single location in DRAM. At application start-up, we reserve 80 MB for the generated code. If the application to be instrumented needs more than that amount of memory for the instrumented code, the instrumentation fail, and our framework cannot be used. Because there is maximal offset for branch and jumps on Cyclops64 we cannot move the instrumented code too far from the original one. This is also why we cannot generate a too big amount of instrumented code.
**Helper thread**

As we can see, in the section about parsing on page 12 we need to parse several times the code get all the information need for the framework to instrument the code correctly, thus the overhead of our system is quiet big. Therefor we decided to use a helper thread instead a normal user thread to instrument the code.

Since we are using an other thread to instrument the code, the Figure 3.4 on page 11 is not completely accurate. Indeed following actions are deported to the helper thread: Code parsing, code generation and instrumentation. So in under this configuration while the helper thread will parse the code, generate code and instrument user’s code, the user’s thread, can run the normal code. This present the nice advantage of completely hiding (timely speaking) the overhead of the framework. The only overhead remaining is that we will use one thread more than the normal application.

**iCache invalidation**

But because the code is not supposed to change during run-time, the hardware instruction cache (iCache) of Cyclops64 is not coherent. This is a big issue for us because that mean that if the code we decide to instrument is being executed at the same time, it will remain in the instruction cache, and the instrumented code will never be executed.

Cyclops64 has a instruction that allow one thread to invalidate its own iCache (ICI: instruction cache invalidate). But we cannot use it because it is an privileged instruction, just kernel code use it. And we don’t really want to use it because first of all, invalidating all the iCache is very costly. Moreover the helper thread cannot use ICI to invalidate the iCache of user’s thread, indeed, when a thread unit (TU) uses ICI it invalidate its own iCache, and the iCache of a thread unit is shared between 10 of them but we have no way to know one which physical TU the logical thread is running.

Therefor we use a software method to invalidate the iCache. As seen in section 3.1, Cyclops64 has an 8 way associative iCache, each of it 64 lines can contain 8 instruction (32bits). Thus to invalidate one single line of the iCache, we can jump eight times to eight wisely chosen instructions, in order to produce 8 cache miss in a row an so, flush the instruction from the iCache. To do that, we use a jump table as we can see of Figure 3.5. This branch table is exactly the size of the iCache : $8 \times 16kB = 128kB$. To use this table to invalidate the correct line of the iCache, we need to know the to which line of the iCache, each line of the branch table correspond to. The formula to get the cache line where an instruction at the address addr will go is :

$$\text{line} = (\text{addr} \gg 6) \& 0x3F$$

Thus knowing in which iCache line the instruction we want to flush is and the line where the first line of the branch tale is, we an find to which instruction of the branch table to jump in order to flush an instruction from the iCache.
To return to the code after using the branch table to flush an instruction from the iCache, we use a register that will keep the address where to branch back.

To invalidate the iCache, the code (i.e. branching to the branch table) must be ran on the TU that need to invalidate an instruction from its iCache. Therefor we cannot use this method to flush an instrumented instruction from the instruction cache, but once an instruction has been instrumented, we can use this method to flush it from the iCache.

**Instruction reordering**

An other feature of the framework is its ability to reorder a bit the instruction. As explained previously, during code parsing, we detect dependency between instrumented instruction and next instruction. This feature allow us to optimize the instruction scheduling. Indeed, we can execute some of the independent instruction following the instrumented instruction in the instrumented code and jump a bit further to skip them when we branch back to the original code. This is the reason why we can stop the dependency check as soon as we find a dependence with the
instrumented instruction.

**Code versioning**

And the last important feature of the framework is code versioning, that mean that we can have several version of code and switch between them depending on different condition at run-time. The three basics code version are:

**Original code:** it is the instruction itself with no modification at all.

**Statistics mode:** it is an instrumented version of the code. While an instruction is in this mode, it will collect run time information about performance we can expect if we set it into performance mode.

**Performance code:** when an instruction is in performance mode, that mean it will use the code generated to improve the performance of it. More detail about this mode will given in the section 3.3 on page 17.

It the helper thread that will switch each instruction from one mode to another. The figure 3.6 summarize the different stats in which an instruction can be and how we can switch between each mode. As we can see, the initial during the initial instrumentation, code parsing and generation the helper thread will set every (instrumented) instruction in statistics mode. And the helper thread work after that will just consist in checking the statistics of each instruction and deciding in which mode to put each instruction. If the statistics of an instruction show that it will not benefits from the ”performance mode” it will set it back to the original one, but if the statistics show that the instruction would benefits of the performance mode, then it will decide to set it into performance mode.

![Figure 3.6](image)

Figure 3.6: Overview of the different modes and how the framework switch between them.

Note: It is when we want to switch from one mode to another that we need to invalidate the iCache. But, as explain previously, we cannot invalidate it in original
code mode. Moreover, we do not invalidate it in performance mode. Indeed, in performance mode when to be as fast as possible thus we don’t want to invalidate even a single line of the iCache. And we don’t want to add even two instructions to check a condition to invalidate the iCache either. So the only mode where we invalidate the iCache (just a single line) is during statistics collection. During this phase, to avoid invalidating the line at each execution of the instruction, we check how many times the instruction have been executed and if it has been executed enough, we will invalidate he line of the iCache containing this instruction each time it is executed (waiting for the helper thread to actually change the mode of this instruction).

3.3 Implementations

In this section we describe different libraries implemented using the framework previously described. Those libraries are designed to improve performances of applications using it. They focus on improving performance of memory access of applications.

3.3.1 Software cache

Cyclops64 doesn’t have a hardware data cache and this is the case for most of the new many-core architecture. The reason why those architectures do have a cache are: first off all cache coherency become more and more complicated to maintain when the number of cores increases, and second not having a cache save a lot of space on the chip, having move space for more cores.

Thus the first try we did was to implement a software manged cache. Implementing a software cache for this architecture will allow applications using it to take advantage of spatial and temporal locality of data to speed up memory access.

Software cache presentation

The software cache we deigned is located in scratchpad memory. We choose to set the cache in scratchpad because it allow each core to have its own cache and also because it closest memory to the processor.

The architecture of the cache is described in figure 3.7. As we can see, each line is composed of eight 64bits elements plus a tag that contain the address of the data currently in the line. And the cache is made of 32 lines. Thus the full size of the cache is 2kB.
The cache is direct mapped, that means the each memory location will go to one single cache location, this present the advantage (compared to associative caches) to have a very fast hit time because we just need to check one value to know if the data is in cache or not. But it increases lower a bit the miss number and can allow line competition between loads or stores.

To keep memory consistent more easily, we have decided to implement a write-through cache policy. That means that for each store, we write the value into memory and in the cache, while with a write-back policy we would just write the value to memory when a line is being flushed from the cache.

**Code**  As explained, the cache is direct mapped, with 32 lines of eight 8 Bytes elements. Thus the formula to get the line where an address should go is:

\[(\text{addr} >> 6) \& 0\times1f\]

As the policy is write-through, the cache code is very simple:

1. Check if the instruction is a store
   - If yes, we perform the store to memory first
2. Calculate the line based on the address of the memory operation
3. Check if we have a hit or miss
   - If miss: load the line with data in memory
4. Perform the memory operation from cache line

The Figure 3.8 summarize the behavior of the cache.

**Statistics and decision:**  Before instrumenting the code into *performance mode*, which is in this case will be using the cache, we want to know if each instruction using it will benefit of using the cache (as explain previously). Therefor, we have a different statistic collection and a decision policy for each different implementation. In the case of the cache, we check if each load has a good locality. To do that, we simulate a cache during the statistic phase to check the hit miss ratio of each load (We do not get statistics for stores, we will explain the reason in the next paragraph). More over we don’t want to use the cache if the load is loading from the scratchpad (because we will not get any speed-up), that is why we also check the number of access to DRAM, SRAM or scratchpad. Based on this data statistics, the helper thread will decide to instrument each load into *performance mode* or to set the original instruction back. The policy is simple: if the hit/miss ratio, is superior to 3 (i.e if we have at least 3 hit for one miss) and if at least 10% of the memory access are in SRAM or DRAM, then the load will be set into *performance mode*, otherwise, it will set the instruction back to the original one.
Cache coherency  Because we might not instrument all instruction in the code, we need to be extra careful about the cache coherency. As the cache is write through, as soon as make one load use the cache, we need to set all the stores of the function into caching mode. That let us keep the cache coherent for all the function.

But if the function call an other function with no instruction using the cache (because it is still in statistic or because it a system function not instrumented all), the cache will loose its coherency (because all stores will not be done into the cache). To tackle this problem, when parsing the code, we check if a branch (outside of the function) happen we will insert a piece of code right after it to invalidate completely the cache.

Performances

The basics performance of the software cache located on scratchpad, are: (as a reminder, depending on the contention on the crossbar, a load will take at least 64
cycles)

**Miss:** at least 80 cycles (depending on network contention)

**Hit:** 24 cycles

So for a hit/miss ratio of one third we should get a very nice theoretical speed-up (around 40%). But this is not at all the case. The reason is that one load almost never comes alone, and when a TU issue a memory operation, a load in our case, it will keep executing the next instructions until the value loaded is needed. So if we have two loads in a row (without dependence), without the caching system we will get the two value in just 65 cycles (1 cycle to perform the first load, and 64 to perform the second load and get the value, and during this time the first load will be done to). But we cannot have this behavior with the software cache because of the dependence inside the cache code, we have to ”pay” the 24 cycles for each load. The other reason why we loose performance is that for all stores, we are just slower than the original code as we have to perform the normal operation plus the cache handling.

To avoid the problem of the multiple load in a row we have integrated a pattern detection in the code parsing phase. When the framework detect more than two loads in a row, it will not instrument them because we know that we have no way to be faster than the original code using the caching system.

When a cache miss happen we need to load a full cache line (64 bytes) and store them into scratchpad memory right after, this is where we lose most of the time, to improve performance of the cache, between the loading of the elements from memory and storing them into the cache, we execute the independent instructions following the load (in the original code). Those instruction have been detected during parsing of the code (see section 3.2.3 on page 12). In that configuration, we do not waste the cycles spent to wait the data from memory as we can execute some instruction during that time.

### 3.3.2 Caching system on registers:

Because of the poor performance of the software cache in scratchpad memory, we implemented a toy version of the cache on registers.

The advantage of putting the cache on registers are that it will speed-up the access to the data because we even don’t have to load them from scratchpad, just a move from register will be enough.

But because we are using the unused registers of the function, we cannot have a cache as big as when it is on scratchpad. The version we have implemented uses 2 lines of 4 elements (each element is 64 bits long). The small size of the cache increases line competition between loads, and as we just have 4 element per line, it also degrade the hit/miss ratio of the cache.

Except the location of the cache (and its size), there is not difference between the software cache on registers and the one on scratchpad.
**Performance**  Here are the measured performances of the caching system on registers:

**Cache miss:** 71 cycles  
**Cache hit:** 20 cycles

As we can see, the cache on register is slightly faster than the one on scratchpad. We will give more details about it in the section dedicated to performances comparison and explanation (section 3.5 on page 24).

### 3.3.3 Prefetching

Because all the caching systems present some performance issue that we couldn’t overcome, we decided to try a completely different memory access optimization policy. So we implemented a automatic prefetching system.

**Prefetching system’s presentation**

As the caching system, we use unused registers to perform allow user’s application to use prefetching through our framework.

We implemented a tool that will prefetch data in advance in loops. Thus we just instrument instructions in loops (innermost). For each load in a loop, we try to prefetch the data two iteration in advance. We can have an overview of how the prefetching happen on figure 3.9.

As we can see, data of the next iterations are prefetched into two different registers. And we need at least two iterations to learn the stride of access, this is why for the first iteration (i=0), the two prefetched data are wrong and we perform the normal load. But for next iterations, the data to load are already in the prefetching registers, so we just need to copy the value into the target register. And perform one prefetching two iteration on advance.

So to generate the prefetching code, we need 5 registers that will be used exclusively by one load: two registers for prefetched data, two registers that will contain the address of the prefetch data and one register that will keep the stride. More over we need 4 registers for internal computation, but this four registers can be shared between all instruction in the loop. Thus the register pressure for a loops with nl loads will be (note that the number of stores doesn’t matter): \( nl \times 5 + 4 \).

Because of the big number of registers needed to instrument a load, we cannot instrument more than two loads in a loop (most of the functions do not have more than 12 unused registers).

The prefetching code flow is illustrated on figure 3.10 on page 23: when a load is instrumented to use prefetching, it will check if the data it wants to access is in the prefetching registers (by checking the address registers), if the expected data is not here, it will compute the stride between the last iteration and the current one. And then prefetch the two next iterations based on this stride. When the data is present in the prefetching register, we just move it to the correct register and we prefetch
the next value (two iterations in advance).

If we have a store in an instrumented loop, the store will check the address where it is storing, and if this address is present in the prefetching registers, it will invalidate this register (by adding replacing this address with 0). In that case, when we will load this address again, the prefetching will fail, and we will not use the old value. However, there is no memory consistency check between the different thread. That mean that if a thread change the value of a memory location that is already prefetched by another thread, the thread using prefetching will read the old value at this memory location. We will explain in section 3.4 why it not such as a big problem as it seems to be.

Statistics and decision

Before instrumenting any instruction to use the prefetching system, we get some information to decide if we want to turn the prefetching on for each instruction. As for the caching system, we get number of access to DRAM and SRAM and scratchpad. We also check the regularity of the access pattern of each load. If at
least 10% of the access are in DRAM or SRAM and that the stride between each iteration is constant, we turn the prefetching on.

**Performances**

The performances achieved using the prefetching system are the following: for a right prefetching, we get the value (and perform the next prefetching) in just 16 cycles, while for a wrong prefetching we need 21 cycles to issue the normal load and the 2 prefetching.

### 3.4 Concurrent execution issues

Because all the tools implemented with this framework will duplicate data and keep them in another location, we have to be careful about memory consistency.

On Cyclops64, the memory consistency model is very strong, it is the sequential consistency model, *i.e.* all the TU of a C64 chip will see the operations to the same memory location in the order.

But as there is no consistency memory verification between threads in each of the implementation we provide, we cannot assure a sequential consistency model with our framework. For instance, let's say we are running 3 threads, and just thread T2 will use either the caching system or the prefetching system. Thread T1 will write data to a shared variable, will threads T2 and T3 will, in a loop read the value of this variable. The thread T2 will just get the first and keep it in cache and just keep reading this value, while the thread T3 will see the up to date value after T1 updated the value. This case is illustrated on listing 3.2.
Listing 3.2: Consistency model breaking exemple

\[
\begin{align*}
T_1: & \quad W(a=1) \quad W(a=2) \\
T_2: & \quad R(a:1) \quad R(a:1) \quad R(a:1) \\
T_3: & \quad R(a:1) \quad R(a:1) \quad R(a:2)
\end{align*}
\]

To avoid having unexpected behavior for a program that needs sequential memory consistency to behave correctly, the programmer needs to add synchronization points. In that case, we could make the framework invalidate all value in either prefetching registers or in cache at each synchronization points. But we actually even don’t have to do that because all synchronizations function are not in-lined, thus each synchronization points in a loop will prevent the framework from instrumenting this loops because of the branch outside of the loop’s body (see section 3.2.3 on page 12).

3.5 Performances

In this section we present several results we achieved on different applications we instrumented with the different optimization feature i.e prefetching and caching system.

The results presented here have been obtained on the simulator (FAST). The main purpose of those results are to prove that we can get performance improvement at least on sequential benchmarks (because for parallel benchmarks, the simulator is no accurate enough to really trust the results).

The table 3.1 shows the results we obtained with the different methods. The performance are presented in therm of speed-up, i.e for the instance, the 35% speed-up obtained on dotproduct application with prefetching optimization means mean that when applying our framework to this application, it runs 35% times faster than without.

Table 3.1: Performances of different applications with different optimization

<table>
<thead>
<tr>
<th>Application</th>
<th>Cache on scratchpad</th>
<th>Cache on registers</th>
<th>Prefetching</th>
</tr>
</thead>
<tbody>
<tr>
<td>dotproduct</td>
<td>-18.34%</td>
<td>3.5%</td>
<td>47%</td>
</tr>
<tr>
<td>matrix multiplication</td>
<td></td>
<td></td>
<td>35%</td>
</tr>
</tbody>
</table>

We can explain the poor performances of dotproduct when we are using cache on scratchpad to optimize the application. The table 3.2 presents the time expressed in cycles to perform one iteration of dotproduct on a dotproduct of 512 elements (with the caching system enabled). For instance we can see on this table that 126 iterations need between 140 and 150 cycles to be executed.

This table can be compared with table 3.3 which present the same information about the non optimized dotproduct code.
As we can see, even if the iterations with cache hit are faster than the normal (between 60 and 70 cycles while without caching it takes between 70 and 80), the number and the time (between 140 and 150 cycles for most of them) needed for each iteration with a cache miss destroy all the benefits we had with the hits.

Table 3.2: Distribution of iterations of dotproduct in number of cycles to perform the iteration with caching optimization

<table>
<thead>
<tr>
<th>Time to perform iteration</th>
<th>Nb of iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>60-70</td>
<td>384</td>
</tr>
<tr>
<td>140-150</td>
<td>126</td>
</tr>
<tr>
<td>190-200</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3.3: Distribution of iterations of dotproduct in number of cycles to perform the iteration without optimization

<table>
<thead>
<tr>
<th>Time to perform iteration</th>
<th>Nb of iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>70-80</td>
<td>512</td>
</tr>
</tbody>
</table>

To prove this we tried to use the caching optimization (on scratchpad memory) on a application that will not produce any cache miss, and we achieved a speed-up of 9.18%. The algorithm used is a dotproduct but just accessing 256 times two of the elements of the each vector.

Even if the preferences of the caching system are not as good as expected, we could achieve a nice speed-up on regular application like matrix multiplication and dotproduct calculation using the prefetching system. For instance we optimized the dotproduct application (the most trivial implementation) which achieve 38 flops without the prefetching system will achieve 78 flops by using the prefetching. We can say that it is a nice speed-up especially because we can achieve it without any change of the source code.
Chapter 4

Achievements and future work

4.1 Achievements

As explained before, we designed an expandable framework which will dynamically rewrite the binary executable at run-time. The framework is able to learn from runtime information if it worth it or not to enable optimization of memory access. We also designed two common ways to improve data access performances, a cache based one and a prefetching based one. All of this is purely done by software modification at run-time.

We managed to get good performance improvement using the prefetching system, for instance we have been able to speed-up a dotproduct by 35%. And even if the caching system still suffer from many issues, it doesn’t affect too badly performance of the initial application.

4.2 Future work

Before explaining more in detail the different possible future work, I think it is important to note that binary rewriting is very a powerful method. Indeed we have a complete view of all the binary, so the transformation we can apply to it are very wide.

The first thing to do would be to try this framework on the real chip, and get more results, by trying more applications.

Is as short term it would be very interesting to implement several other cache policy for the caching system. For instance, implementing a write back cache would highly lower the overhead of stores.

It would also be interesting to implement an associative cache. An associative cache would lower the number of miss, and would also tackle the line competition problem.

Because the access pattern of a code portion can greatly change during the time, it would be interesting to put regularly all instructions into learning phase again. Indeed, that will prevent the framework from using performance mode when it is
not worth any more because the access pattern changed, or it would allow to put
some instructions in \textit{performance mode} even if it was not good for them to be in
this mode before. In this case the framework will adapt to new application’s access
pattern.

We also could go deeper into instruction reordering, for instance we should be
able to improve performance of the current implementations by handling two instru-
mented instructions next to each other in the same block. This would leave us more
space for instruction scheduling and would prevent us to jumping twice (once to the
instrumented code, and once to jumps back to the original code).

Also to go deeper into instruction reordering we could do automatic software
pipelining.
Conclusion

4.3 Personal benefits

This internship has been very rewarding for me on several aspects. First the work I realized allowed me to go very deeply into processor architecture. Thus it let me take great advantage of architecture course I had at the University.

Also, working on dynamic rewriting is very interesting, first because it is not common to modify a binary while it is running, and also because to do so we need a complete understanding of the hardware. It is also interesting to how we can change the behavior of a program by doing that, and we need to be extra careful not to change the semantic of the program.

It is also very important to understand completely all the notion learned in system’s classes to understand, how to change the binary without modifying its semantic. For instance how to do a function call, or how to use the function’s stack.

Another aspect of the internship that fulfilled my expectation, even if it is not completely dependent of the internship itself, is that it allowed me to improve mt English’s skills. Because I spent 6 months in the US, it let me time to talk to local people in daily life and to have more technical discussion in the University and in the team, allowing me to improve both my technical and daily life English.

Finally, this internship comforted me in the will of continuing my education by doing a PhD. I really enjoyed working in CAPSL as a research scholar.

4.4 Acknowledgment

I would like to thanks especially Joseph Manzano and Jean Christophe Beyler for there help, there pedagogy, and there stupid questions. And of course there quiet impressive technical knowledge.

I also really appreciate the handful of times we had meetings in Iron Hills, about my work or to help deciding where do my PhD.

I want to thank Joseph for all his very helpful advises and especially about presenting and writing.

I believe that if this experience has been as rewarding for me, Joseph and Jean Christophe are responsible of it for a very important part.
Bibliography

