LIGHT-PREM: AUTOMATICALLY TRANSFORMING EXISTING APPLICATIONS TO THE PREDICTABLE EXECUTION MODEL

BY

ROMAN DUDKO

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Adviser:

Professor Marco Caccamo

Abstract

As real-time embedded systems become more complex, there is a desire to use these systems on higher performance commercial off-the-shelf (COTS) components. However, worst case execution times (WCET) are unreasonably high on these components, due to contention amongst resources. Past work has introduced the Predictable Execution Model (PREM) to solve this issue, but unfortunately, the time required to port existing code bases to this model is too high. Light-PREM is a tool that aims to extend this framework by automating certain processes that previously had to be done by hand. This thesis goes over the interface, implementation and preliminary results of two different implementations of Light-PREM.

Acknowledgments

I would like to thank many people who aided me in my research, without whom this thesis would not have been possible. I would like to thank my advisor, Professor Marco Caccamo, for giving me guidance and support throughout my research and during the thesis process. I would also like to thank Gang Yao for assisting me in developing the Light-PREM Source Analyzer, and providing consultation. I would like to thank the research group involved in creating the Predictable Execution Model (PREM), which this research builds off of. These people include Professor Marco Caccamo, Rodolfo Pellizzoni, Emilliano Betti, Stanley Bak, and Gang Yao, amongst other people. I would like to thank Renato Mancuso for providing ideas for different approaches to Light-PREM. Finally, I would like to thank my Mother, Eugenia Chudinova, my Father, Yuriy Dudko, my little sister, Natalie Dudko, my girlfriend, Lara Troszak, and my friends, for their continued love and support, and for helping to relieve stress on tough weeks.

Table of Contents

Chapter 1

Introduction

Real-time embedded systems are becoming increasingly more complex and have higher performance requirements. Current custom made hardware components used in real-time systems give good guarantees on worst-case execution times, but often fall behind in performance from commercial off-the-shelf (COTS) components by several orders of magnitude. As such, there is a desire to use COTS components to improve the performance of real-time embedded systems, but unfortunately COTS components are not designed with real-time systems in mind. COTS systems are highly optimized for the average case, but execution times in the worst-case can grow to unreasonable size, due to contention between multiple resources, such as the CPU, bus, memory, I/O etc. To consider a specific case, if multiple I/O peripherals were to attempt to access the bus at the same time, many of the I/O peripherals would be waiting for the other peripherals to finish using the bus before they could use it, which would increase the execution time, even though no useful work was done.

Past work has focused on solving this problem, and introduced a new execution model called the Predictable Execution Model (PREM) [16]. PREM aims to reduce worst case execution times (WCET) by using a high-level co-scheduling mechanism of peripherals and the CPU. Using PREM allows real-time systems to use COTS components and gain the advantage of higher performance without suffering the disadvantage of large WCETs. However, the process to modify an existing program to the PREM model is time-consuming, and arguably infeasible for larger code bases. This work aims to extend the PREM framework, by giving real-time system developers a tool to help them automatically port existing applications to the PREM model. This tool, in combination with the PREM framework, is referred to as *Light-PREM*.

1.1 Thesis Overview

This thesis begins with an overview of background knowledge, by briefly reviewing the PREM model as discussed in [16], and draws attention to the issue this thesis aims to address. The reader is instructed to refer to the original paper for a more detailed explanation of PREM. Section 1.3 continues by describing the goals of Light-PREM, the interface of the tool, and briefly describing two different approaches of Light-PREM. Section 1.4 goes over related work about automatically prefetching memory.

Following the introduction, the two main contributions of this thesis are discussed in two seperate chapters. Chapter 2 explains the idea behind the first implementation of Light-PREM, discusses advantages and disadvantages to the approach, gives instructions on how to use the tool, goes over the implementation, and concludes with experimental results. Chaper 3 explains the idea behind

the second implementation of Light-PREM, and similarly discusses advantages, disadvantages, instructions, implementation and experimental results.

Finally, this thesis provides insight into possible directions for future work in Chapter 4, and finishes with concluding remarks in Chapter 5. Included at the end of this thesis, Appendix A lists important figures and charts, and Appendix B lists code related to Light-PREM.

1.2 PREM

The PREM model focuses on resolving contention on the bus and main memory by co-scheduling DMA peripherals and the CPU together. Figure 1.1, borrowed from the original PREM paper [16], succinctly describes how this is done. Real-time bridges sit between all peripherals and the main bus. The job of real-time bridges is to buffer all traffic from the peripherals, and only deliver it once the scheduler allows it. All real-time bridges communicate with a peripheral scheduler, which schedules all of the real-time bridges so that only one peripheral can access the bus, and only when the schedule allows it. Managing peripheral traffic resolves contention for the bus and main memory amongst peripherals [1], but there is also the CPU to consider.

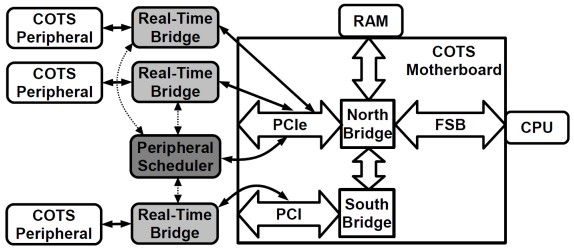


Figure 1.1: Real-Time I/O Management System

The CPU is scheduled in combination with the peripheral scheduler in such a way so that there is zero contention for the bus. To understand how this is done, the system model first needs to be explained. As in traditional real-time scheduling theory, a task set of a CPU Γ consists of N periodic tasks, such that $\Gamma = \{\tau_1, \tau_2, ..., \tau_N\}$. Peripherals have a similar task set, defined as $\Gamma^{I/O} = \{\tau_1^{I/O}, \tau_2^{I/O}, ..., \tau_M^{I/O}\}$. Each CPU task τ_i however, is split into N_i scheduling intervals $\{s_{i,1}, s_{i,2}, ..., s_{i,N_i}\}$. Thus, the timing requirements of a task τ_i can be expressed as $\{\{e_{i,1}, e_{i,2}, ..., e_{i,N_i}\}, p_i, D_i\}$, where $e_{i,j}$ is the WCET of scheduling interval $s_{i,j}$, p_i is the period of the task, and D_i is the relative deadline. Scheduling intervals are split into two types: compatible intervals and predictable intervals.

Compatible intervals are execution intervals that have no constraints, and can make system calls, access main memory, etc. As such, no peripheral traffic is allowed to communicate on the bus during these intervals, since there could be some contention. For this reason, compatible intervals should also be kept as small as possible.

Predictable intervals are split into two phases, the memory phase and the execution phase. During the memory phase, the process prefetches all needed cache lines, so that they reside in the cache, and no more accesses to main memory will be needed later during this interval. After the memory phase, the execution phase occurs, where the process performs its tasks. The memory phase

exists so that this execution phase does not need to access the bus, which allows peripheral traffic to communicate over the bus concurrently during this time. As such, during the execution phase, no system calls are allowed, because these might access main memory. Similarly, no preemptions are allowed, as other processes could kick out of the cache some of the needed cache lines. The execution phase should take into account the WCET, however should the process finish early, the execution phase should still busy wait until the phase is over, so that peripherals still have a chance to communicate over the bus. A good diagram depicting the memory and execution phase is shown in Figure 1.2, again borrowed from [16]. All tasks are assigned priority, with CPU tasks always having higher priority than I/O tasks, and are scheduled using the Rate-Monotonic (RM) scheduling algorithm [8], with a slight modification to take into account that tasks cannot be preempted.

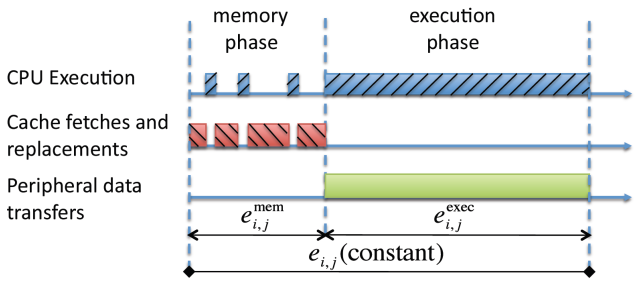


Figure 1.2: Predictable Interval with constant execution time

Thus, in order to move a real-time system to the PREM framework, the following steps must be performed first:

1. A real-time bridge must be interposed between peripherals
2. A peripheral scheduler must be added, and
3. CPU tasks must be transformed so that they are split into scheduling intervals, which must either conform to the requirements of a compatible interval or predictable interval.

Since there is only one peripheral scheduler, it is a one-time cost. Similarly, since a system generally only has a couple peripherals that need to be added, adding real-time bridges is a reasonable requirement. Writing drivers for real-time bridges can be tough, but thankfully some Linux drivers can be reused for this purpose. However, in the current model of PREM, transforming CPU tasks (step 3) requires that the programmer manually divides tasks into scheduling intervals, and manually adds prefetch statements for all memory that is needed in the execution phase of predictable intervals. This roughly requires performing the following process:

1. Profile the code, and split tasks into appropriate scheduling intervals
2. For each scheduling interval, add macros (defined by the PREM library) to demark the beginning and end of that scheduling interval
3. For predictable intervals, add prefetch statements in the memory phase by:
   (a) Running the code through a memory analyzing tool such as callgrind [23]
   (b) Find variables causing cache misses highlighted by this tool

(c) For all variables that cause cache misses, find their size in memory, by manually examining/understanding the source code.

(d) Add a prefetch statement in the memory phase section, including the variable name and its size.

(e) Return to step a. until no cache misses occur.

As the size of the source code increases, so does the time required to manually perform this process. As such, this method is not scalable in terms of the size of the code base. Light-PREM aims to automate all or most of this process. Currently, this thesis shows two different implementation of Light-PREM that assume steps 1 and 2 have already been performed, and automatically performs step 3, provided the programmer provides some minimal information such as how to compile the source code. Thus the current focus of Light-PREM is on automatically generating code for predictable intervals.

1.3 Light-PREM

1.3.1 Goals and Restrictions

In developing Light-PREM, we had some goals set forth. As mentioned in the previous section, we aimed to automate the process of converting source code to the PREM model, and in particular, focused on adding prefetch statements for predictable intervals first. The most complete solution to add these prefetch statements would be to operate at the compiler level, since the compiler already parses the source code, and provides data structures such as the Abstract Syntax Tree (AST) which would allow much easier manipulation of the program. We looked into utilizing compiler passes such as those offered by the LLVM [6] compiler framework to automatically add prefetch statements at compile time. However, this method would violate another goal we had established, which was to remain compiler-independent. Since many developers use their own compiler, some of which may be custom-made, using a specific compiler to implement Light-PREM would be unreasonable for them; they would be unwilling to switch compilers. Thus, we aim to implement Light-PREM at the source code level instead. This technique may not always yield perfect results, since we are not able to analyze the code as well as a compiler can. The goal is to be able to handle most cases, and have the programmer fix the rest, or accept the fact that there will be some contention, but significantly less. Past work shows how the WCET time can be adjusted in this case, to account for minor peripheral interferences that the task may encounter [17, 18]. Finally, due to the fact that PREM is currently implemented in C, Light-PREM only aims to convert programs written in C.

As mentioned previously, Light-PREM assumes that a predictable interval has already been identified, and two macros have already been added to demark the predictable interval. Typically this will look like:

```
void myPredictableInterval()
{
        // beginning of predictable interval
        PREM_BEGIN

        ...
        // original code
        ...

        // end of predictable interval
        PREM_END
}
```

After Light-PREM is finished, it will generate code that looks like this:

```
void myPredictableInterval()
{
        // beginning of memory phase, beginning of predictable interval
        PREM_BEGIN_PRED_INTERVAL(0);

        // Light-PREM generated prefetch statements
        PREM_PREFETCH_GLOBAL(&x, sizeof(x));
        PREM_PREFETCH_GLOBAL(malloced_array, 20 * sizeof(int));
        ...

        // end of memory phase, beginning of execution phase
        PREM_BEGIN_PRED_EXECUTION("myPredictableInterval", 0);

        ...
        // original code
        ...

        // end of execution phase, end of predictable interval
        PREM_END_PRED_INTERVAL("myPredictableInterval", 0, 0);
}
```

For more detailed examples, we refer the reader to appendix B.1, which shows examples from actual runs of Light-PREM on existing code.

1.3.2 Approaches

This thesis shows two different implementations of Light-PREM, which are called Light-PREM Source Analyzer and Light-PREM Memory Analyzer respectively. Each implementation is entirely separate from the other, and takes different approaches to determine what memory needs to be prefetched. They can be combined together to achieve the most desirable results. The first method that was developed was Light-PREM Source Analyzer, which aimed to automate the process humans performed already, and is more focused on parsing source code using regular expressions. To mitigate certain disadvantages of this approach, Light-PREM Memory Analyzer was developed second, and is more focused on analyzing memory usage patterns during runtime. The two different

implementations thus get their names from what they focus on. Both implementations are discussed in this thesis. In discussing both implementations, the terms "user", "programmer" and "developer" refer to clients of the Light-PREM program, i.e. people that use Light-PREM to convert their own existing programs to the PREM model.

1.4 Other Related Work

The Light-PREM tool needs to prefetch memory in predictable intervals, and it does this with the aid of memory profilers and memory tracers. There is a wealth of existing work in both prefetching memory and memory profilers/tracers.

Due to the fact that processor speeds have been increasing exponentially in past years, as according to Moore's law, the gap between the performance of memory subsystems and processors has increasingly grown wider. To address this issue, much research has focused on improving cache performance. One common approach to improving cache hit rates is prefetching memory asynchronously shortly before it is needed, thus reducing memory stall times.

Light-PREM, similarly, aims to prefetch memory, however for much different reasons, and with different constraints. Whereas past research has used memory prefetching to increase performance (and thus improve the average case execution times), the PREM model uses prefetching to eliminate contention to memory, reducing worst case execution times. These different goals lead to other differences, such as the fact that most memory prefetching strategies uses compiler techniques, whereas Light-PREM aims to remain compiler independent. Also, Light-PREM places prefetch statements at the beginning of an execution interval, whereas existing memory prefetching research places prefetch statements throughout the execution interval. Due to these major differences, Light-PREM's prefetching strategy is orthogonal to most existing memory prefetching research, however it is still useful to consider this existing body of work because of the types of analyses performed. That is, while the end goal may be different, the analysis techniques used to determine data usage patterns is shared.

Existing work in memory prefetching can be split into two different areas: software based and hardware based. Light-PREM must generate prefetch statements in source code, so only software based approaches are reviewed here. One such approach is a compiler algorithm that uses locality analysis to selectively prefetch references that are most likely to cause cache misses [13]. This algorithm is most effective on dense matrix computations, typically found in scientific engineering applications. This algorithm was later extended in [12] to perform well on multiprocessor platforms as well. The same author of these algorithms also developed a memory prefetching algorithm that excels on pointer-based applications and applications using recursive data types [9, 10]. The authors of these papers described three distinct strategies for doing this, which were greedy prefetching, history-pointer prefetching and data-linearization prefetching. Greedy prefetching prefetches all neighbor pointers, history-pointer prefetching construct history-pointers, which are used in subsequent runs for prefetching, and data-linearization involves rearranging memory to take advantage of spatial locality. These methods are compiler-based, and as mentioned previously, Light-PREM aims to remain compiler independent. However these strategies could be generalized to be compiler-independent with some work.

Other memory prefetching strategies include prefetching function parameters that are point-

ers [7], using jump pointers to prefetch memory in pointer-based structures [20], prefetching data reference sequences that frequently repeat (called "hot data streams") [2], and exploiting stride-based memory accesses [4, 11, 26], which are memory accesses where the difference between 2 successive accesses remains relatively constant during some execution interval.

Existing memory prefetching strategies like the ones described above generally require prefetch statements either be added by hand, or are added automatically, by using a compiler or profiler to aid in the analysis. Light-PREM similarly must use a memory profiler and memory tracer to perform its functions. The current memory profiler/tracer that Light-PREM uses is Valgrind [14], however other memory profilers and tracers exist as well.

When considering memory profilers and tracers, it is important to note the difference between the two. Memory profilers analyze the frequency of program instructions and usage, whereas memory tracers provide a complete listing of all data references. Light-PREM is mostly focused on memory tracing, because it needs to determine which memory was accessed and thus needs to be prefetched. Some examples of memory tracers include QPT [5] and Pin [19]. Memory tracing generally involves using existing compiler techniques such as Reaching-Definition analysis to determine which statements contribute to calculating a memory address, so that the memory address can be recomputed by the memory tracer and printed/stored [5]. A good overview of past and present memory tracers and how they trace memory can be found in [5].

Although Light-PREM aims to remain compiler-independent, one compiler tool that could safely disregard this restriction is the ROSE compiler infrastructure [21], which can perform source-to-source transformations. Since this compiler framework generates source code instead of a binary file, users can use their own compiler after the transformation is complete.

Chapter 2

Light-PREM Source Analyzer

In this chapter, whenever I refer to Light-PREM, I am specifically referring to Light-PREM Source Analyzer.

2.1 Explanation

Light-PREM Source Analyzer aims to automate the process humans perform when generating prefetch statements for predictable intervals. This process is outlined at the end of section 1.2, specifically step 3. For the ease of the reader, step 3 is listed again, here:

1. Running the code through a memory analyzing tool such as callgrind [23]
2. Find variables causing cache misses highlighted by this tool
3. For all variables that cause cache misses, find their size in memory, by manually examining/understanding the source code.
4. Add a prefetch statement in the memory phase section, including the variable name and its size.
5. Return to step 1 until no cache misses occur.

The process that Light-PREM performs can be mapped to the process listed above, and is best explained in this manner:

1. Light-PREM is able to perform step 1 by simply calling an existing profiling tool called callgrind. This tool indicates which lines in the source code suffered cache misses.
2. To perform step 2, Light-PREM can utilize the information provided by callgrind; however, since callgrind only indicates which lines suffer cache misses, and not specifically which variables, Light-PREM must try to guess which variable(s) on those lines caused the cache miss. Instead of guessing, Light-PREM uses a regular expression to pull out all possible variable names, and tries to prefetch all of them. It is possible that Light-PREM may prefetch variables that did not cause cache misses, but this is ok, since prefetching something that is already in cache does not negatively affect the program.
3. In step 3, Light-PREM must try to determine the size of the variable it is trying to prefetch. In most cases, if you are given a variable `x`, it is sufficient to simply prefetch `sizeof(x)` bytes, thus generating a prefetch statement as such:

`PREM_PREFETCH_GLOBAL(&x, sizeof(x));`

However, in the case of pointers and arrays, this is not sufficient. Thus, Light-PREM aims to determine the size of arrays allocated by `malloc` and arrays declared statically. It does this by examining the source code and using regular expressions to find `malloc` calls and statically declared arrays. Light-PREM can then create a mapping from array variable names to their appropriate size in bytes. For arrays allocated by `malloc`, this is simply the size passed to `malloc`, and for statically allocated arrays, it is sufficient to use the `sizeof` operator. For example, for a statically declared array `a`, a prefetch statement might look like:

`PREM_PREFETCH_GLOBAL(a, sizeof(a));`

Note that the first argument to the macro above is `a` and not `&a`.

4. After having found all variables, and their sizes, step 4 is relatively easy. Light-PREM inserts prefetch statements into the code, making sure not to prefetch the same variable twice, and using the size of the variable determined from step 3 above to determine how much to prefetch.

5. To return to step 1, the user can simply call Light-PREM again.

2.1.1 Advantages

Light-PREM Source Analyzer is able to handle simple programs fairly well, as shown in section 2.4. Compared to Light-PREM Memory Analyzer, Light-PREM Source Analyzer is able to prefetch statically declared arrays much more effectively, and is several orders of magnitudes faster at performing profiling and analysis. Due to its simplicity, it is easier to understand, modify and maintain.

2.1.2 Disadvantages

Light-PREM Source Analyzer is not able to handle more complex programs very well however, due to an issue known as aliasing [25]. Aliasing refers to the principle that the same memory location may be accessed through different symbolic names. Aliasing by itself is not an issue, but rather the issue stems from the fact that aliased names may be out of scope in the memory phase section, and it is difficult to determine mappings between aliased names. For example, consider the following very simple code:

```
void foo(int *a, int size)
{
        for (int i = 0; i < size; i++)
                a[i] = 2*i;
}

void myPredictableInterval ()
{
        int array[] = {1,2,3,4,5};
        foo(array, 5);
}
```

In this example, Light-PREM would try to prefetch the integer pointer `a`, however `a` is out of scope in the function `myPredictableInterval`. Instead we want to prefetch the variable `array`, however it is not trivial in the general case to determine the mapping from `array` to `a`. Light-PREM Memory Analyzer aims to avoid this problem, and is described in the next chapter.

2.2 Instructions for Use

This section lists instructions on how to use Light-PREM Source Analyzer to convert a function named `foo` into a predictable interval. These instructions assume that Light-PREM Source Analyzer is included on the machine's `PATH` variable. Note that some of these steps are required to setup PREM for the target project. If PREM is already setup and enabled, you may skip steps 1-3.

1. Include the compiled PREM object `prem_support.o` in the linking stage of your executable.
2. Enable PREM support by defining `PREM_ENABLE` and `PREM_PATCH`. This can be done by adding the following compilation flags: `-DPREM_ENABLE -DPREM_PATCH`
3. At the top of the file containing the function `foo`, add the line: `#include "prem_support.h"`
4. Copy the sample configuration file `sample.cfg` from the Light-PREM config directory into the target project directory. Uncomment lines in this file and set configuration values appropriately. Some parameters such as the name of the predictable function are required. See appendix B.2.1 for an example configuration file.
5. (Optional) Rename `sample.cfg` to `light.cfg`.
6. Add the following lines to the top of the function `foo`:

   ```
   PREM_BEGIN_PRED_INTERVAL(0);
   PREM_BEGIN_PRED_EXECUTION("foo", 0);
   ```

   Also add the following line to the bottom of the function `foo`:

   ```
   PREM_END_PRED_INTERVAL("foo", 0, 0);
   ```

7. Run "`light_prem light.cfg`" from the target project directory.

2.3 Implementation

This section describes in detail how Light-PREM Source Analyzer is implemented. It is written in C++ and bash scripts. In converting source code to PREM, Light-PREM executes several (5) stages, as listed below. These steps are roughly equivalent to the 5 stages seen in the Light-PREM Source Analyzer script listed in appendix B.2.2.

2.3.1 Configuration

Light-PREM first loads all of the configuration parameters from the specified configuration file. If any required parameters are omitted, Light-PREM notifies the user what is missing, and aborts. A sample configuration file can be found in appendix B.2.1.

2.3.2 Formatting

In order to ease analysis of the source code, it is first run through the C preprocessor and formatted. Since Light-PREM performs analysis on the source code itself, this ensures that, for example, if the programmer puts curly braces on a new line or on the same line as an `if` statement, Light-PREM will be able to analyze the code regardless. Light-PREM still keeps a copy of the original un-formatted code however, to which it adds prefetch statements, so developers using the tool do not have to worry about the formatting of their code changing. The C preprocessor used was gcc's preprocessor, and the code formatter that was used was uncrustify [22], a fairly configurable formatter that matched the needs of Light-PREM well. The format that we configured uncrustify to format the source code to can be found appendix B.2.4. This formatting allowed us to make assumptions about the structure of the code later on. In particular, regular expressions to pull information out of the source code became much easier to write.

Note that in using the gcc preprocessor, we did not violate staying compiler-independent, because we only use gcc as a tool to aid in the analysis. This means that developers can still use their own compiler to later compile their programs into the PREM version.

2.3.3 Memory Initializations

The next step that Light-PREM performs is to find all `malloc` calls and static array declarations, so that a mapping from array name to size can be created, as required in step 3 listed in the Explanation section (section 2.1). This is done by compiling the program, using a makefile provided by the user, and then running callgrind [23] on the executable. Here we use callgrind to determine which lines in the source code were called before the predictable interval was called (indicating a memory initialization that occurred before the predictable interval started). Using the results of callgrind, Light-PREM finds all lines in the source code that are either a static array declaration, or a `malloc` call, and that were executed at least once. To determine that a line is a static array declaration, the following regular expressions are used:

```
([a-zA-Z_][a-zA-Z0-9_]*)[ \t]*[[][]]
^([a-zA-Z_][a-zA-Z0-9_]*[ \t])*([a-zA-Z_][a-zA-Z0-9_]*)[ \t]*[[].*[]][ \t]*;
```

To determine that a line is a `malloc` call, the following regular expression is used:

```
(.*)=.*malloc[^(]*[(](.*)[)]
```

The reader will notice that these regular expressions contain capture groups (expressions surrounded by parentheses), which are used to pull out the array names, and in the case of heap-allocated arrays, their size as well. The names of these variables, and associated sizes, are recorded in temporary files for future use.

2.3.4 Cache Misses

Next, Light-PREM reruns callgrind, but in a different mode. In this run, Light-PREM uses callgrind to record execution only during the predictable interval. After doing this, Light-PREM analyzes the output of callgrind to find lines on which cache misses occur. However, just provided the line number is not enough to exactly determine which variable caused the cache miss, since there may be multiple variable on one line. Thus, Light-PREM pulls out all variables, using the following regular expression:

```
([a-zA-Z_][a-zA-Z0-9_]*)[ ]*[^( a-zA-Z0-9_]
```

Note that the character range `[a-zA-Z0-9_]` denotes the characters that are allowed in a variable name as defined by the C language. Since numbers are not allowed as the first character in a variable name, the first character must be in the range `[a-zA-Z_]`. The first parenthesis, formed around the group `[a-zA-Z_][a-zA-Z0-9_]*`, denotes a capture group, which captures the name of the variable. The rest of the regular expression is needed to ensure that the entire variable name is captured, and not just part of it.

As an example, take the following line of source code: `array[i] = x + foo(y);`. The above regular expression would be able to pull out the following variable names: `array`, `i`, `x`, `y`. Note that the function `foo` was not included in this output, since it is a function. The regular expression avoids pulling out function names.

Once all variables names are extracted, they are first checked to see if they belong to the list of variables that were allocated using `malloc`. If the variable is, it is prefetched, using the size that was passed into `malloc`. However, if the variable is in the list of variables that were declared as a static array, Light-PREM prefetches it, using the `sizeof` operator to find the size of the array. Finally, if the variable is in neither list, it is simply prefetched. The source code that performs these steps can be found in appendix B.2.3.

2.3.5 Extraneous Prefetches

In extracting all variables from lines with cache misses, some variables will be unnecessary. In general, it is ok to prefetch more than is needed, but if the variable is not in scope in the place where we add prefetch statements, then the code will not compile. This is similar to the issue described in section 2.1.2. This is the reason for this last step. The code is run through the compiler, and if a prefetch statement does not compile (because the compiler cannot find the given variable name), then the prefetch statement is simply removed.

2.4 Experimental Results

In order to test the effectiveness of Light-PREM, it was run on 7 benchmarks from the EEMBC [3] benchmark suite. In past work [16], the benchmarks were modified to run multiple times, instead of just once, so that timing could be more accurately measured. The benchmarks in these experiments were run with 10,000 iterations, except for bitmnp, which was run with 1,000 iterations, due to its longer execution time. Each benchmark was run 3 times, one for each of the following scenarios:

1. No modifications
2. Light-PREM modifications
3. Manual (by hand) modifications

The results of these experiments are shown in Figures A.1-A.7 in appendix A.1. The 7 benchmarks that were tested were a2time, basefp, bitmnp, cacheb, canrdr, rspeed and tblook. Each benchmark has two bar graphs, one depicting the cache misses suffered during the predictable interval (only during the execution phase) and another depicting the number of bytes that were prefetched in the memory phase. The cache misses were measured using the script found in appendix B.2.5.

The manual (by hand) version always has the best results, since a human can do much more detailed analysis, at the cost of time. It may be noted however, that the manual version was not able to eliminate all cache misses. This artifact was noted in previous work [15], and is attributed to the fact that the cache replacement policy on the testbed that we used was random, meaning there was a chance a previously fetched cache line was evicted by another cache line. In caches with a Least Recently Used (LRU) cache replacement policy, we expect this artifact to disappear.

In all 7 benchmarks, Light-PREM was not able to identify all variables that needed to be prefetched. The reason for this is that one particular variable, which reoccurred in all 7 benchmarks, could not be associated with its allocated memory. This variable was named `RAMfilePtr`, and it was used as a pointer into an allocated array named `RAMfile`. Light-PREM was able to associate `RAMfile` with its size, since the variable `RAMfile` was assigned the return value of a `malloc` call. However, Light-PREM was not able to determine that `RAMfilePtr` was associated with that same memory. This again refers to the issue of aliasing described in section 2.1.2. When Light-PREM saw a cache miss on the `RAMfilePtr` variable, it simply prefetched the first element from this array. Thus, the code was structured like so:

```
int *RAMfile, *RAMfilePtr;

int main()
{
        // Light-PREM associates RAMfile with 500*sizeof(int)
        RAMfile = malloc(500 * sizeof(int));

        // Light-PREM knows nothing about the size of RAMfilePtr
        RAMfilePtr = RAMfile;

        myPredictableInterval();
}

void myPredictableInterval()
{
        for (int i = 0; i < 500; i++)
        {
                // cache miss on RAMfilePtr
                int value = *RAMfilePtr;
                RAMfilePtr++;
                ...
        }
}
```

In some of the benchmarks, this code structure outlined above occurred for more than one variable, such as in cacheb and tblook. When these specific variables were added in by hand, Light-PREM was verified to have the same amount of cache misses as the manual version, and prefetched less bytes than the manual version. This behavior is able to be detected by Light-PREM Memory Analyzer, which is described in the next chapter.

The acute reader will notice another set of listed Experimental Results in appendix A.2 related to the JPEG benchmark. This benchmark is much more complicated than the EEMBC benchmarks, and Light-PREM was ineffective at generating prefetch statements in these benchmarks, due to the issues listed in section 2.1.2. For this reason, Light-PREM Source Analyzer was intentionally omitted from these results.

Chapter 3

Light-PREM Memory Analyzer

In this chapter, whenever I refer to Light-PREM, I am specifically referring to Light-PREM Memory Analyzer.

3.1 Explanation

3.1.1 Concept

Where Light-PREM Source Analyzer had trouble prefetching memory, Light-PREM Memory Analyzer aims to succeed. As discussed in section 2.1.2, Light-PREM Source Analyzer had trouble with name aliasing. Whereas a human might have been able to solve name aliasing issues by examining the source code, Light-PREM Source Analyzer does not have this complex analysis available. Instead, Light-PREM Memory Analyzer aims to avoid this issue altogether by taking a different approach to the problem: by observing memory usage patterns during runtime.

Light-PREM Memory Analyzer uses a subtool of Valgrind called Lackey, which prints to standard error the absolute address location of all memory accesses of a program during execution. Light-PREM aims to connect this information with local and global variables available to the predictable interval within its scope. By doing so, Light-PREM can convert absolute addresses, which change from run to run, to relative values based off of variable names. Connecting memory accesses to variables is the main problem Light-PREM Memory Analyzer aims to solve. Once this is done, generating prefetch statements is trivial.

Let us again consider the example from section 2.1.2, which Light-PREM Source Analyzer was not able to handle. For the ease of the reader, it is listed here again:

```
void foo(int *a, int size)
{
        for (int i = 0; i < size; i++)
                a[i] = 2*i;
}

void myPredictableInterval()
{
        int array[] = {1,2,3,4,5};
        foo(array, 5);
}
```

Suppose that the memory tracer, Lackey, tells us that during the lifetime of the function

`myPredictableInterval`, it accessed memory locations `0xabcd0000` through `0xabcd0014`. Light-PREM can print the address of `array`, and discover that it points to memory location `0xabcd0000`. Then it is trivial for Light-PREM to map the memory accesses of `0xabcd0000` through `0xabcd0014` to `array+0` through `array+5`, and it can then easily prefetch the variable `array` for `sizeof(int)*5` bytes.

However, it is not so trivial when mapping memory accesses to variable names involves following pointers. Consider another simple example:

```
void foo(int *a, int size)
{
        for (int i = 0; i < size; i++)
                a[i] = 2*i;
}

void myPredictableInterval (struct struct_type *some_struct)
{
        foo(some_struct->array, 5);
}
```

In this example, Lackey might report memory accesses at addresses `0xabcd0000` through `0xabcd0014`, but this time `some_struct` will have an address in some other address chunk, maybe `0xffff1234`. If Light-PREM was able to understand the type of `some_struct`, and could examine the `struct` fields of `some_struct`, it would eventually be able to find the pointer `array` inside `some_struct`, and use that pointer to prefetch the memory accesses. However, understanding `struct` types, or any type for that matter, is a task best left to the compiler. In order to avoid delving into such complicated logic, we instead observe the fact that `some_struct->array` is equivalent to `*(some_struct+C)`, where `C` is some constant. All Light-PREM has to do is find this `C`. One method that Light-PREM could use to achieve this is such a probing algorithm:

```
int x = 0;
while (*(void**)((void*) some_struct + x) != 0xabcd0000)
        x++;
```

Note that this same probing algorithm can also be applied to arrays of pointers, because an array access such as `array[i]` can be equivalently expressed as `*(array+C)`, where `C` happens to equal `i`.

This probing algorithm [1] will work in the case of the example, but in general, we run the risk that eventually we might probe too far, and try to access invalid memory. The solution to this problem is to find out what memory is valid, and what memory is not.

To do this, Light-PREM intercepts `malloc` calls, and stores information about which address ranges are valid heap memory address ranges. Since most pointers point to heap memory, and most of the memory that we want to prefetch is on the heap anyway, this is sufficient to tell us whether or not an address is valid. In future work, we hope to also incorporate the stack address ranges.

[1] Some like to call this algorithm "fishing for pointers" instead

3.1.2 Strategy

Having understood the general idea behind Light-PREM Memory Analyzer, it is now possible to describe more concisely the analysis it performs without getting lost in the details.

Given a function, Light-PREM needs to prefetch all memory addresses that that function accesses. Inside this function, the only variables it has access to at the very beginning of its lifetime is its own function parameters, and global variables. All memory accesses can be expressed using these variables as a starting point. We refer to these variables as "handles". Light-PREM must connect handles with the absolute memory addresses that Lackey reports in order to successfully prefetch accessed memory. In the last example, `some_struct` was a handle, since it was a function parameter, and it needed to be connected with the memory address `0xabcd0000`. This "connection" would look like `*(some_struct+C)`, where `C` would be some constant integer.

As previously mentioned, Light-PREM keeps track of all heap memory address ranges (from now on referred to as heap chunks). By treating heap chunks as just opaque data, Light-PREM can perform the probing algorithm on the heap chunks themselves, and determine how the heap chunks are connected. Pseudocode for how this can be done is listed below. The full algorithm is listed in appendix B.3.2.

```
int pointerSize = sizeof(void*);
int n = num_heap_chunks;
for (int i = 0; i < n; i++)
        for (void* ptr = chunk[i].start; ptr + pointerSize < chunk[i].end; ptr++)
                void *value = *((void**)ptr);
                for (int j = 0; j < n; j++)
                        if (isBetween(value, chunk[j].start, chunk[j].end)
                                // make connection from heap chunk i to j.
```

Figure 3.1 shown below depicts an example scenario, where, after intercepting `malloc` calls, Light-PREM knows about 3 different heap chunks. It also knows about a handle variable named `handle1` which has address `0xaaaa0001`. By performing the probing algorithm, Light-PREM determined that 2 bytes into heap chunk 1, there are some bytes that can be treated as a pointer into heap chunk 2. Other connections are also depicted.

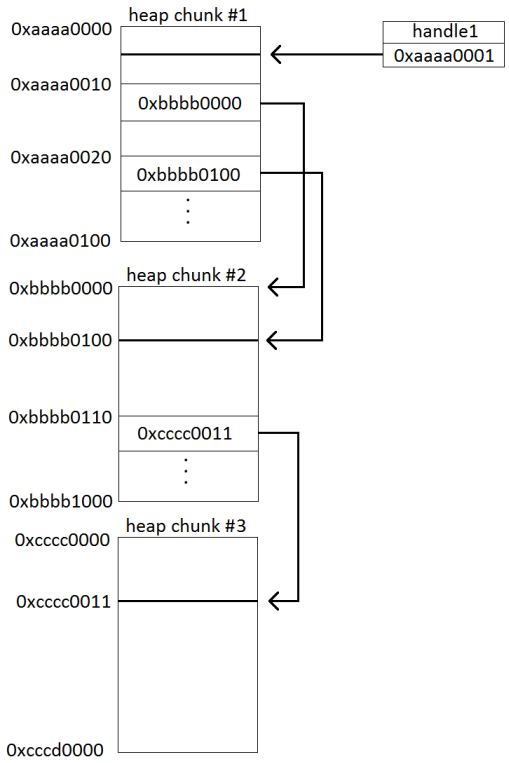


Figure 3.1: Layout of Heap Memory in Example Scenario

Figure 3.1 can thus also be interpreted as a directed multi-graph, and alternatively be drawn as shown in figure 3.2. Heap chunks are represented by circles, and handles are represented by rectangles. Connections between heap chunks are represented by two numbers. To understand this notation, consider heap chunk `a` and heap chunk `b`, where `a` and `b` denote the start address to each respective chunk. We mark the edge from heap chunk `a` to heap chunk `b` as `+x, -y` if:

```
*(void**)((void*)a + x) - y == (void*) b
```

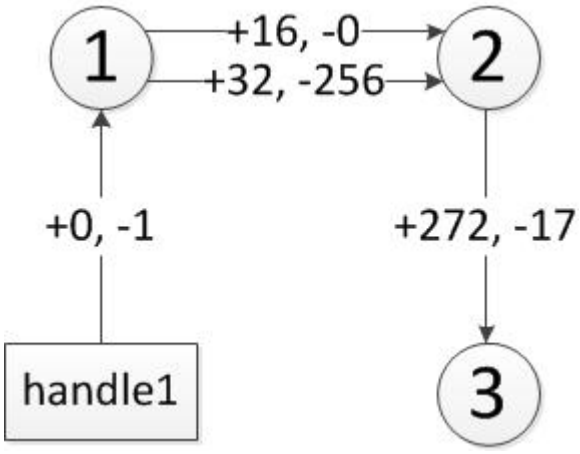


Figure 3.2: Graph representing connections amongst heap chunks from figure 3.1

Consider now that Lackey reports a memory access at address `0xcccc0101`. This address belongs to chunk 3. According to the graph, there exists a path to this address from a handle, by traversing from handle1 to heap chunk 1 to heap chunk 2 and finally to heap chunk 3. Thus, to prefetch `0xcccc0101`, Light-PREM would prefetch the memory associated with:

```
*(void**)(*(void**)(((void*)handle1) + 15)  + 272) + 254
```

This is the main idea behind Light-PREM Memory Analyzer. It constructs a graph of the heap memory, making edges between chunks a and b if there exists a pointer in a that references some address in b. Light-PREM then uses these edges to connect handles with reported memory accesses, and then prefetches these expressions. For an additional example, please refer to appendix B.1.6, which shows a complete and simple example program, and the Light-PREM generated prefetch statements for it.

3.1.3 Advantages

Light-PREM Memory Analyzer is able to completely avoid the name aliasing problem, and thus is more effective than Light-PREM Source Analyzer, and is able to handle much more complicated source codes. This is backed up by Experimental Results listed in section 3.4.

3.1.4 Disadvantages

Light-PREM uses the Valgrind tool Lackey, which tends to be extremely slow when profiling memory accesses. A simple program that only takes a few seconds to execute outside of the profiling environment may take tens of hours to complete in the profiling environment. This can grow out of proportion for even larger programs.

Light-PREM also has an issue that it cannot follow pointers through the stack. The graph shown in the previous section only listed connections amongst heap chunks, and not between heap chunks and the stack. This is due to the fact that Light-PREM only keeps track of heap chunks. Thus a pointer that resides on the stack is considered as invalid memory by Light-PREM and not followed. This can result in various memory accesses not being prefetched.

In performing the probing algorithm to determine connections between heap chunks, Light-PREM may happen to come across some data that does not represent a pointer, but yet coincidentally has a value that could be interpreted as an address in a valid memory region. The solution to this problem is to run the heap analysis twice. Since memory addresses change from run to run, it is unlikely that this coincidence would occur again in the second run.

Finally, the code that Light-PREM produces cannot be 100% guaranteed to be safe, and in the context of real-time and embedded systems, this factor is very important. There is always the possibility that an assumption that was made about the memory access patterns under one input is not valid against a different input. This could result in invalid memory accesses, potentially leading to corrupt memory and segmentation faults. We leave it to future work to address these issues. It is also important to keep in mind that Light-PREM is only meant to be used as a tool to help generate prefetch statements. The user can eliminate prefetch statements afterwards that he deems unsafe or unnecessary.

3.2 Instruction for Use

This section lists instructions on how to use Light-PREM Memory Analyzer to convert a function named `foo` into a predictable interval. These instructions assume that Light-PREM Memory Analyzer is included on the machine's `PATH` variable. Note that some of these steps are required to setup PREM for the target project. If PREM is already setup and enabled, you may skip steps 1-3.

1. Include the compiled PREM object `prem_support.o` in the linking stage of your executable.
2. Enable PREM support by defining `PREM_ENABLE` and `PREM_PATCH`. This can be done by adding the following compilation flags: `-DPREM_ENABLE -DPREM_PATCH`
3. At the top of the file containing the function `foo`, add the line: `#include "prem_support.h"`
4. At the top of the file containing the function `foo`, add the line: `#include "lightprem.h"`
5. Create a file named `lightprem.h`, and place the lines `#define PREM_BEGIN` and `#define PREM_END` inside of this file.
6. Add the macro `PREM_BEGIN` in the first line of the function `foo`, and place the macro `PREM_END` in the last line of the function `foo`
7. Add the heap analyzer to the linker, by including the file `lib_lightprem_malloc.so` in the list of objects in the linking stage of compilation.
8. Inside of your Makefile, make sure to include `lightprem.h` in the list of file dependencies.
9. Export a variable named `PREPROCESSOR_OPTS` to tell Light-PREM what preprocessor options are needed to successfully run the preprocessor on the file containing `foo`. Example: `export PREPROCESSOR_OPTS="-I ../prem"`.
10. Run "`lightprem source func exe exe_args`", where `source` is the name of the file containing the function `foo`, `func` is the name of the predictable function (here it would be "foo"), `exe` is the name of the executable, and `exe_args` are the arguments for the executable.
11. Finally, remove `lib_lightprem_malloc.so` from the linker.

3.3 Implementation

This section describes in detail how Light-PREM Memory Analyzer is implemented. It is written in C, C++ and bash scripts. In converting source code to PREM, Light-PREM executes several (5) stages, as listed below. These steps are roughly equivalent to the 5 stages seen in the Light-PREM Memory Analyzer script listed in appendix B.3.1.

3.3.1 Formatting

Light-PREM needs to determine the names of all handles, which as was mentioned in section 3.1.2, is either a global variable or a function parameter. This requires examining the source code. Thus, just like Light-PREM Source Analyzer, Light-PREM Memory Analyzer must convert the code to a

specific format so that it can then use regular expressions to pull out the names of all handles. This is the only information that Light-PREM needs to parse out of the source code. Refer to section 2.3.2 for a more detailed explanation of the Formatting process.

3.3.2 Profiling Code Generation

After pulling out all handle variable names, Light-PREM generates profiling code to print out the addresses of the handles, and to also call a function called `dump_mem_graph`. This function originates from the shared object file included by Light-PREM (see section 3.2, step 7). This shared object file, also known as the heap analyzer, keeps track of all heap memory address ranges by intercepting `malloc` calls. When `dump_mem_graph` is called, it performs the probing algorithm as described in section 3.1.2, and then dumps to a file a list of all heap chunks, followed by a list of heap connections, specified one per line. The code that performs this process can be found in appendix B.3.2.

3.3.3 Comparison Run

Light-PREM runs the target executable twice, to avoid a problem listed in section 3.1.4. Specifically, since the probing algorithm treats heap chunks as opaque data, a piece of data may coincidentally happen to have a value that can be interpreted as a memory address into a valid memory region. In order to avoid this, the heap analysis is run twice. This step, labeled "Comparison Run", is the first of two of such runs. In the next step, labeled "Profile Run", the heap analysis is run again, and connections that are not present in both analyses are ignored.

3.3.4 Profile Run

Light-PREM finally runs the target executable under Lackey, in order to get a list of all memory accesses. At the same time, the heap analysis code, included into the executable via a shared object, is also run at the same time. At the end of this step, two files are produced. The file `lightprem.info` contains the address ranges of all heap chunks, followed by the connections that were discovered between them, and finally followed by a list of all of the memory accesses, expressed as 8 digit hexadecimal values, one per line. The file `vars.info` contains a list of all handle variable names, and their values at the beginning of the predictable interval. At this point, Light-PREM has all the information it needs. It passes this information to the last stage, which generates the actual prefetch statements.

3.3.5 PREM Code Generation

At this point, Light-PREM has a list of all memory accesses, a list of all handle variables and their values, and a list of all the heap chunks and their connections. Light-PREM gives this information to the Prefetch Generator. The Prefetch Generator first constructs a graph like the one shown in figure 3.2. For the ease of the reader, it is shown again below as figure 3.3. Light-PREM uses a custom-built C++ graph class to represent this graph. After reading the input and creating this graph, the Prefetch Generator takes every memory access, and attempts to find a path from a handle variable to the heap chunk that that memory access resides in.

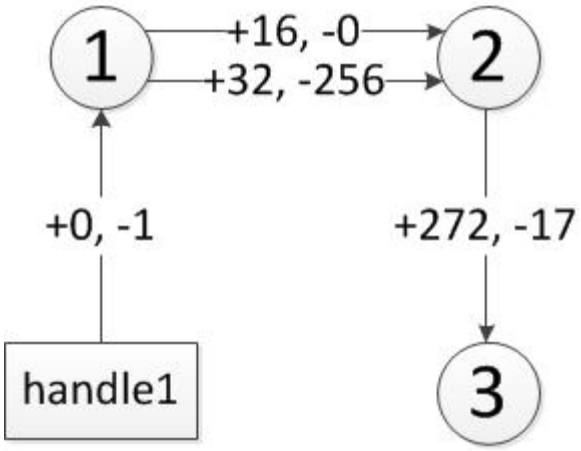


Figure 3.3: Graph representing connections amongst heap chunks from figure 3.1

In order to find such a path, the Prefetch Generator performs a reverse depth-first search. That is, it inverts the direction of all edges, and performs a depth-first search starting at the heap chunk that the memory access resided in, stopping when it finally reaches a handle node (depicted as rectangles in the figure). In performing the depth-first search, the Prefetch Generator does not (intentionally) remember which nodes it visited, and since the graph could be cyclic, this means that potentially the Prefetch Generator could get into an infinite loop. To prevent this, the Prefetch Generator only searches up to a predefined maximum depth, currently set to 10.

The reason for this unusual graph traversal is two-fold: We want to keep pointer offsets small, and we need to ensure that pointer offsets are non-negative. It is for these reasons that the Prefetch Generator uses a depth-first search instead of a breadth-first search, and does not (intentionally) remember visited nodes.

Firstly, the reason we want to keep pointer offsets small is because these pointer references are much more likely to be valid. Consider another graph, as show below.

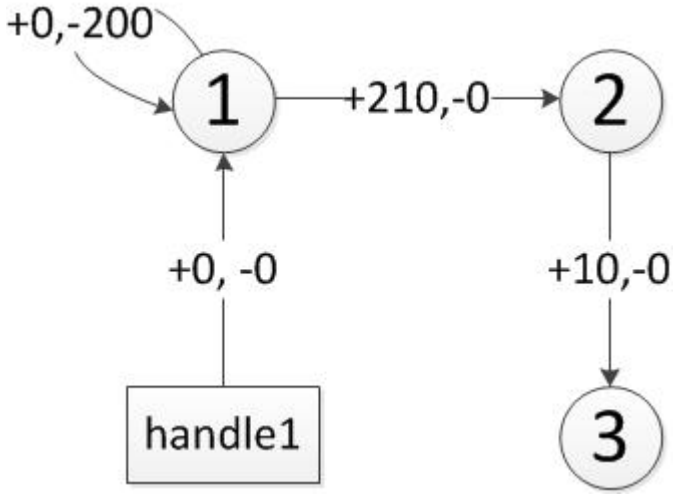


Figure 3.4: Heap Memory Graph with a self-loop

This time, the graph contains a self-loop. That is, heap chunk 1 has a pointer 0 bytes inside of it, that points to an address 200 bytes inside of it. If we were trying to find a path to heap chunk 2, we would prefer to take the self-loop before going to heap chunk 2. This is because of what these paths represent:

```
Path with    self-loop: *(void**)(*(void**)((void*) handle1 + 0) + 10)
Path without self-loop: *(void**)((void*) handle1 + 210)
```

It is much more likely that `handle1 + 0` is a valid pointer than `handle1 + 210`. This is a heuristic, but it is fairly effective. The reason that `handle1 + 210` may be invalid is because heap chunk 1 may contain other data inside of it, some of which may be variable sized. Thus 210 may not be a constant offset, depending on what data resides in the heap chunk. However, pointers that are closer together are much more likely to belong to the same structure or array, and thus the offset is more likely to be constant. This scenario is depicted below in figure 3.5.

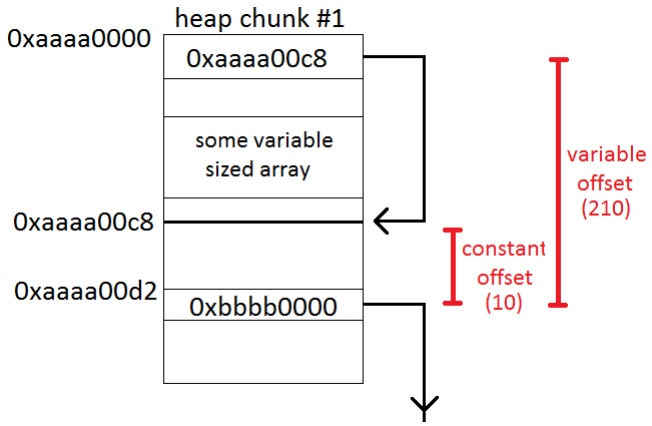


Figure 3.5: Example Layout of Heap Memory based on graph from figure 3.4

Secondly, we want to ensure that pointer offsets are always positive. Let us consider three kinds of pointers (in C): structure pointers, array pointers, and normal pointers. We mentioned previously that `some_struct->field` could be equivalently expressed as `*(some_struct+C)`, that `some_array[i]` could be equivalently expressed as `*(some_array+i)` and that `*some_pointer` could be equivalently expressed as `*(some_pointer + 0)`. In all cases, the offset is non-negative, thus we want to make sure that when traversing the graph, the Prefetch Generator also ensures that pointer offsets are non-negative. Thus, the Prefetch Generator does not consider non-negative offsets, but instead ignores them.

Once the Prefetch Generator has found a path from a handle to a memory access, it prints out the prefetch statement that represents this path. The Prefetch Generator also aggregates prefetch statements at the end, so that statements like:

```
PREFETCH(array+0, 4);
PREFETCH(array+4, 4);
PREFETCH(array+8, 4);
PREFETCH(array+12, 4);
```

Get combined into a single statement:

```
PREFETCH(array+0, 16);
```

The source code for the Prefetch Generator can be found in appendix B.3.3, and the C++ class that was used to represent the graph can be found in appendix B.3.4.

3.4 Experimental Results

Light-PREM Memory Analyzer was run on the original EEMBC benchmark suite that Light-PREM Source Analyzer was run on, and additionally, it was also run on a JPEG benchmark suite. The results are described here.

3.4.1 EEMBC

In order to test the effectiveness of Light-PREM, it was run on 7 benchmarks from the EEMBC [3] suite. In past work [16], the benchmarks were modified to run multiple times, instead of just once, so that timing could be more accurately measured. The benchmarks in these experiments were run with 10,000 iterations, except for bitmnp, which was run with 1,000 iterations, due to its longer execution time. The results of these experiments are shown in Figures A.1-A.7 in appendix A.1. The 7 benchmarks that were tested were a2time, basefp, bitmnp, cacheb, canrdr, rspeed and tblook. Each benchmark has two bar graphs, one depicting the cache misses suffered during the predictable interval (only during the execution phase) and another depicting the number of bytes that were prefetched in the memory phase. The cache misses were measured using the script found in appendix B.2.5.

Since Light-PREM Source Analyzer already compared the effectiveness of itself with the manual and Non-PREM versions depicted in these graphs, I will instead focus on the differences between Light-PREM Memory Analyzer and Light-PREM Source Analyzer. For a more detailed discussion in comparison to the Non-PREM and manual versions, please see section 2.4.

Light-PREM Memory Analyzer overall performed significantly better than Light-PREM Source Analyzer, with 5 out of the 7 benchmarks suffering over 50% less cache misses. Light-PREM Memory Analyzer performed worse on the benchmark bitmnp, but this is most likely due to the fact that it was not able to retrieve statically declared arrays, which are not located on the heap, but instead in their own memory section. However, since heap-allocated data was the cause for the majority of cache misses on the other benchmarks, Light-PREM Memory Analyzer was more successful because it focuses on heap data.

Light-PREM Memory Analyzer, while being more effective, could still be combined with Light-PREM Source Analyzer to even further improve performance. While the graphs are not included, it was verified that combining the two strategies results in cache misses equal to exactly that of the manual version. Once Light-PREM Memory Analyzer is improved to handle other kinds of memory chunks, not just heap chunks (which we leave to future work), we expect that Light-PREM Memory Analyzer will be able to prefetch everything Light-PREM Source Analyzer could, and much more.

3.4.2 JPEG

To test Light-PREM on a larger, more complicated code base, we ran Light-PREM Memory Analyzer on the JPEG Image Encoding Benchmark. As described in [16], the JPEG benchmark was discovered to use over 80% of execution time in a function called `compress_data`. This function became the target for Light-PREM Memory Analyzer to turn into a predictable interval.

During analysis of the JPEG code, it was discovered that one pointer in particular went through the stack, which as described in section 3.1.4, is a limitation of the Light-PREM Memory Analyzer that it, as of currently, cannot handle. Thus, the JPEG source code was slightly modified, moving this single pointer on to the heap, so that Light-PREM Memory Analyzer could effectively follow this pointer. We leave it to future work for Light-PREM to handle this case without having to modify the source code. After having done this, Light-PREM Memory Analyzer was run with two different input files. One input file represented a 512x512 pixel image, and another input file represented a 3456x2304 pixel image. These files were converted from a .ppm format to a .jpg format during the benchmark. Note that the second image has over 30 times more pixels that the first image.

The prefetch statements that were generated by Light-PREM on these two different sized input files were mostly the same, except for some prefetches that were based on the image width. By comparing the two sets of prefetch statements, and doing a simple analysis of the differences, we were able to replace the integer constants corresponding to the image width with a variable that represented the image width. Remember that Light-PREM is meant to be a tool to help users in creating prefetch statements. In our opinion, it is reasonable for the users of Light-PREM to go back and look at the prefetch statements, and make some minor adjustments.

Having performed these modifications, Light-PREM performed excellently, reducing cache misses by over 94% in both the 512x512 image and the 3456x2304 image. These results can be seen in figures A.8 and A.9. The number of bytes prefetched by Light-PREM was also less than that of the manual version, indicating that Light-PREM was probably able to avoid prefetching data that was already in the cache or was unused. Light-PREM was still not able to match the performance of the manual version in the larger image, suffering roughly twice as many cache misses. However this difference is much less significant when compared to the number of cache misses with and without Light-PREM.

Chapter 4

Future Work

Light-PREM Memory Analyzer takes an approach that is much more effective at prefetching memory than Light-PREM Source Analyzer, thus future work should be devoted to improving Light-PREM Memory Analyzer first. There is still much that Light-PREM Memory Analyzer could be improved in, whereas further improvement in Light-PREM Source Analyzer is limited because of its approach and design. Thus I will focus on future work for Light-PREM Memory Analyzer, and when I refer to Light-PREM, I am specifically referring to Light-PREM Memory Analyzer.

As was discussed in the Disadvantages section (section 3.1.4) of Light-PREM Memory Analyzer, Light-PREM is not able to follow pointers that do not go through the heap section. Other such memory sections include the stack section, data section, and BSS section. Future work should look into including these sections in the Light-PREM analysis.

Another issue that was discussed in the Disadvantages section is that Light-PREM cannot be 100% guaranteed to generate prefetch statements that will not cause segmentation faults. Currently, we are looking in to writing a custom segmentation fault handler, which ignores segmentation faults on prefetch statements, and instead skips to the next prefetch statement. This would incur additional runtime overhead, and is not ideal, but it can at least guarantee safety, which is critical in real-time and embedded systems. We would expect this segmentation fault handler to be called rarely.

Light-PREM uses a subtool of Valgrind called Lackey, which as was discussed earlier, is extremely slow. Future work could look into writing a custom version of Lackey, or improving it, or using a different memory access profiler altogether. The documentation of Lackey explicitly states that it emphasizes "clarity of implementation over performance" [24]. Most likely, there is a lot of room for performance improvements in the Lackey tool.

Section 3.2 lists the steps required to run Light-PREM, and as of currently, there are a total of 11 steps. While these steps do not take long to perform, it would be fairly easy to automate these steps. Future work could look into streamlining this process.

Finally, there are different avenues that Light-PREM could explore to enhance its performance. Some possible ideas are to lock memory in the cache in the granularity of pages, having users annotate their source code to allow Light-PREM to more easily analyze the code, and introducing specific programming rules on what is allowed in a predictable interval. We will be looking into these directions in the future.

As was mentioned in the end of the PREM section (section 1.2), Light-PREM also assumes that the programmer has already profiled the code to split it into scheduling intervals, and has added macros to demark the beginning and end of scheduling intervals. Future work should look into automating this process as well.

Chapter 5

Conclusions

The Predictable Execution Model (PREM) is a novel framework to move real-time embedded systems onto COTS components, so that higher performance can be achieved, while still keeping worst case execution times (WCET) down. However, porting existing applications to this model requires significant code transformations, which places a heavy burden on the programmer. In particular, tasks such as adding prefetch statements during the memory phase of predictable intervals is very time consuming.

Light-PREM aims to extend PREM by automatically performing this task. Currently two implementations of Light-PREM have been developed, which show promising results. These two implementations are called Light-PREM Source Analyzer and Light-PREM Memory Analyzer.

Light-PREM Source Analyzer uses existing profiling tools to detect cache misses, and uses regular expressions on formatted source code to extract possible variables to prefetch. While this version of Light-PREM shows good results on small benchmarks such as the EEMBC benchmark suite, it does not perform as well on more complicated code bases, due to name aliasing of variables.

Light-PREM Memory Analyzer aims to be more flexible and robust than Light-PREM Source Analyzer, by taking a different approach that avoids name aliasing problems. In particular, it uses a memory access profiler to detect potential cache misses, and then observes memory usage patterns during runtime in order to figure out how to access these memory addresses, in the hopes of prefetching them in the memory phase. Currently, Light-PREM Memory Analyzer is slow, and is only able to observe memory usage patterns on the heap, but future work will aim to resolve these issues. Despite significant disadvantages to the approach, which will ideally be handled in future work, Light-PREM Memory Analyzer still performs very well, prefetching over 94% of cache misses on a complicated and large code base: the JPEG Image Encoding Benchmark.

Clearly Light-PREM has proven that it can achieve the task of automatically generating prefetch statements for the memory phase of predictable intervals, while still remaining compiler independent, thereby reducing the transformation effort required to move an application to the PREM framework. Hopefully this means that using real-time embedded systems on COTS components will become easier and easier over time as PREM and Light-PREM continue to improve.

References

[1] S. Bak, E. Betti, R. Pellizzoni, M. Caccamo, and Lui Sha. Real-time control of i/o cots peripherals for embedded systems. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 193 –203, dec. 2009.

[2] Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 199–209, New York, NY, USA, 2002. ACM.

[3] EEMBC. Eembc. http://www.eembc.org/, 2012.

[4] Tatsushi Inagaki, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Stride prefetching by dynamically inspecting objects. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 269–277, New York, NY, USA, 2003. ACM.

[5] J.R. Larus. Efficient program tracing. *Computer*, 26(5):52 –61, may 1993.

[6] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75 – 86, march 2004.

[7] Mikko H. Lipasti, William J. Schmidt, Steven R. Kunkel, and Robert R. Roediger. Spaid: software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th annual international symposium on Microarchitecture*, MICRO 28, pages 231–236, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[8] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[9] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. *SIGPLAN Not.*, 31(9):222–233, September 1996.

[10] Chi-Keung Luk and Todd C. Mowry. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Transactions on Computers*, 48, 1999.

[11] Chi-Keung Luk, Robert Muth, Harish Patil, Richard Weiss, P. Geoffrey Lowney, and Robert Cohn. Profile-guided post-link stride prefetching. In *Proceedings of the 16th international conference on Supercomputing*, ICS '02, pages 167–178, New York, NY, USA, 2002. ACM.

[12] Todd C. Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *ACM Trans. Comput. Syst.*, 16(1):55–92, February 1998.

[13] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, ASPLOS-V, pages 62–73, New York, NY, USA, 1992. ACM.

[14] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[15] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, and M. Caccamo. Predictable execution model: concept and implementation. 2010.

[16] R. Pellizzoni, E. Betti, S. Bak, Gang Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 269 –279, april 2011.

[17] R. Pellizzoni and M. Caccamo. Impact of peripheral-processor interference on wcet analysis of real-time embedded systems. *Computers, IEEE Transactions on*, 59(3):400 –415, march 2010.

[18] R. Pellizzoni and M. Caccamo. Impact of peripheral-processor interference on wcet analysis of real-time embedded systems. *Computers, IEEE Transactions on*, 59(3):400–415, 2010.

[19] Pin. Pin - a dynamic binary instrumentation tool. http://www.pintool.org/, 2012.

[20] A. Roth and G.S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*, pages 111 –121, 1999.

[21] Quan Sun and Hui Tian. The ROSE source-to-source compiler infrastructure. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT*, Galveston Island, Texas, USA, October 2011.

[22] Uncrustify. Uncrustify. http://uncrustify.sourceforge.net/, 2012.

[23] Valgrind. Callgrind. http://www.valgrind.org/docs/manual/cl-manual.html, 2012.

[24] Valgrind. Lackey. http://www.valgrind.org/docs/manual/lk-manual.html, 2012.

[25] Wikipedia. Aliasing (computing). http://en.wikipedia.org/wiki/Aliasing_(computing), 2012.

[26] Youfeng Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 210–221, New York, NY, USA, 2002. ACM.

Appendix A

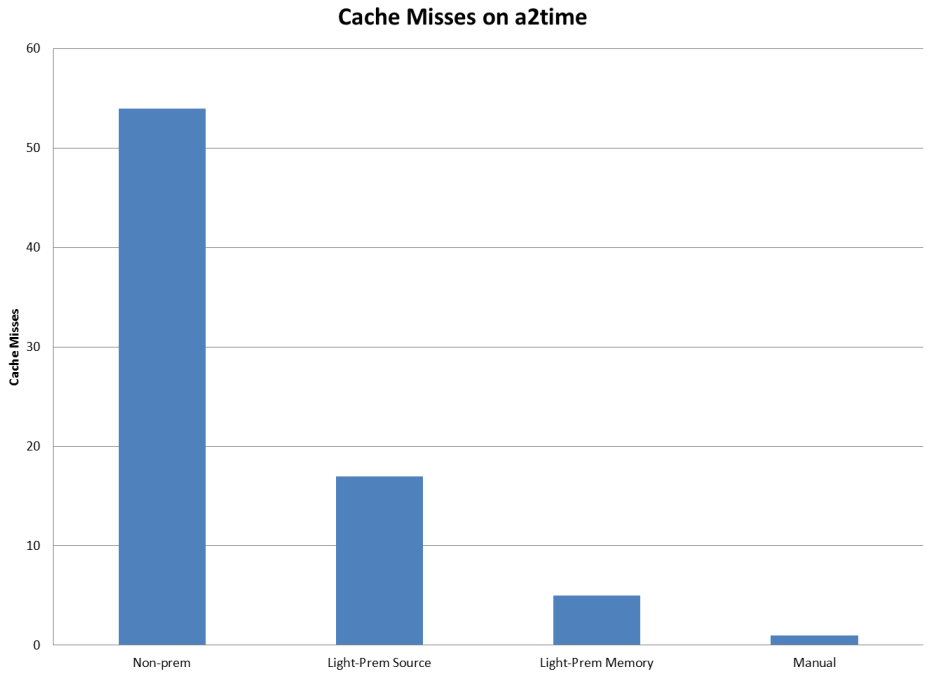
Figures

This appendix shows figures depicting Experimental Results from both versions of Light-PREM.

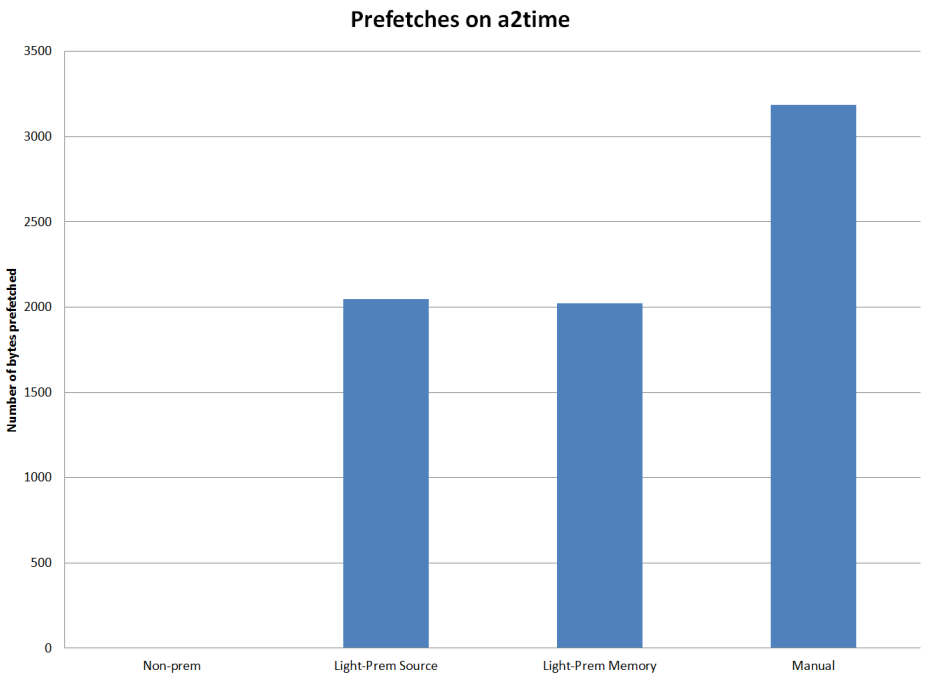
A.1 EEMBC Experimental Results

The following figures show the number of cache misses and the number of prefetched bytes on the EEMBC benchmarks after having generated code using Light-PREM.

A.1.1 a2time



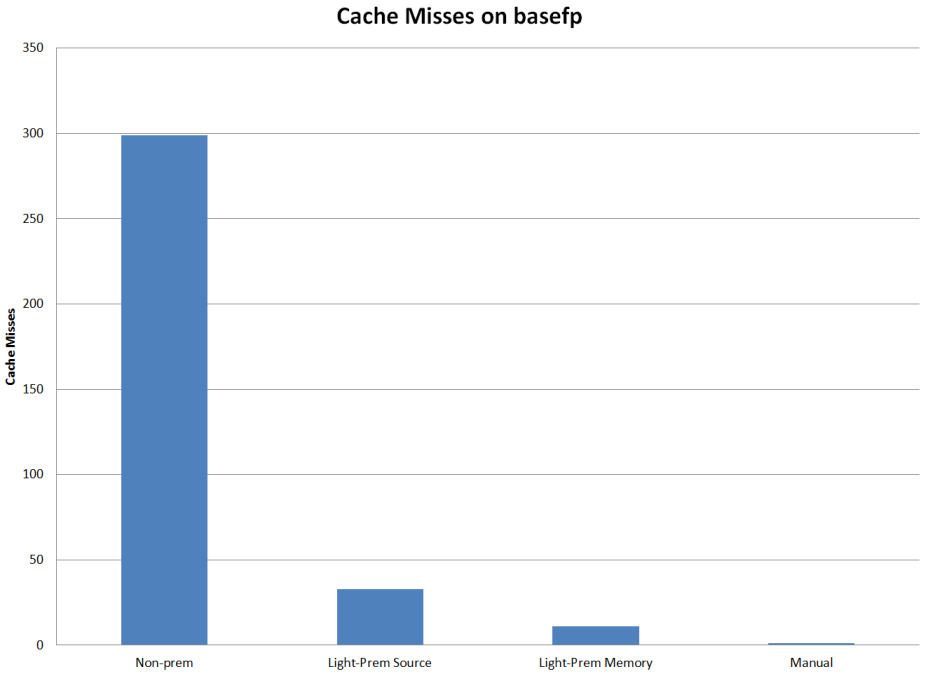
(a) Cache Misses



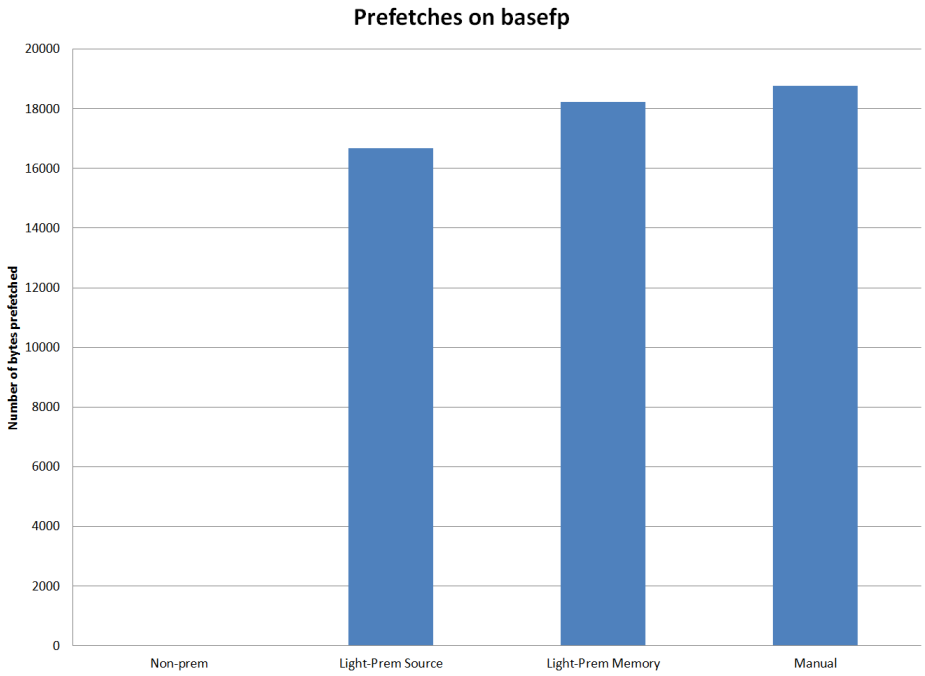
(b) Prefetches

Figure A.1: Cache Misses and Prefetches on a2time

A.1.2 basefp



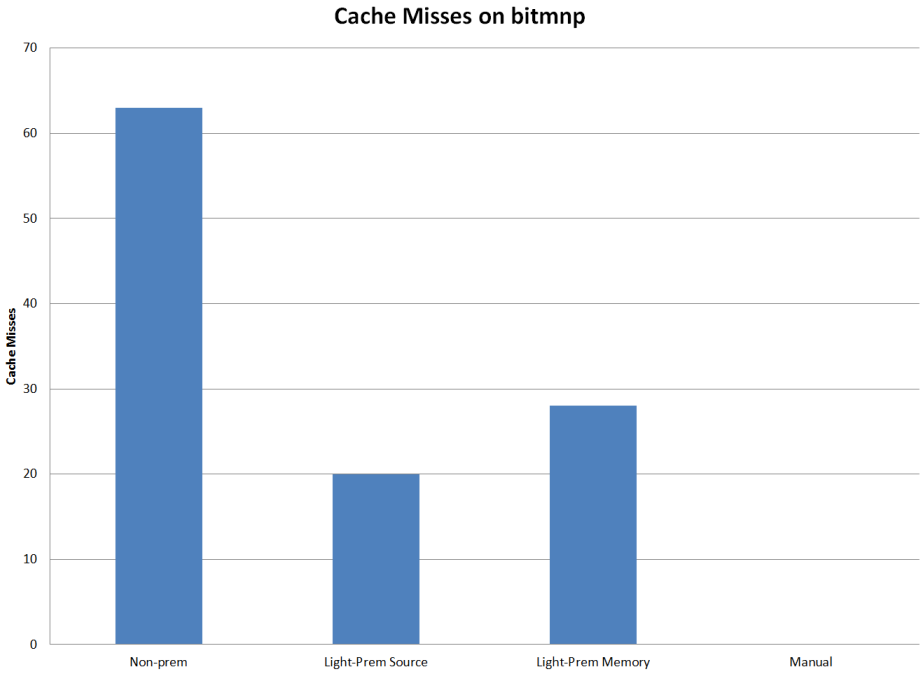
(a) Cache Misses



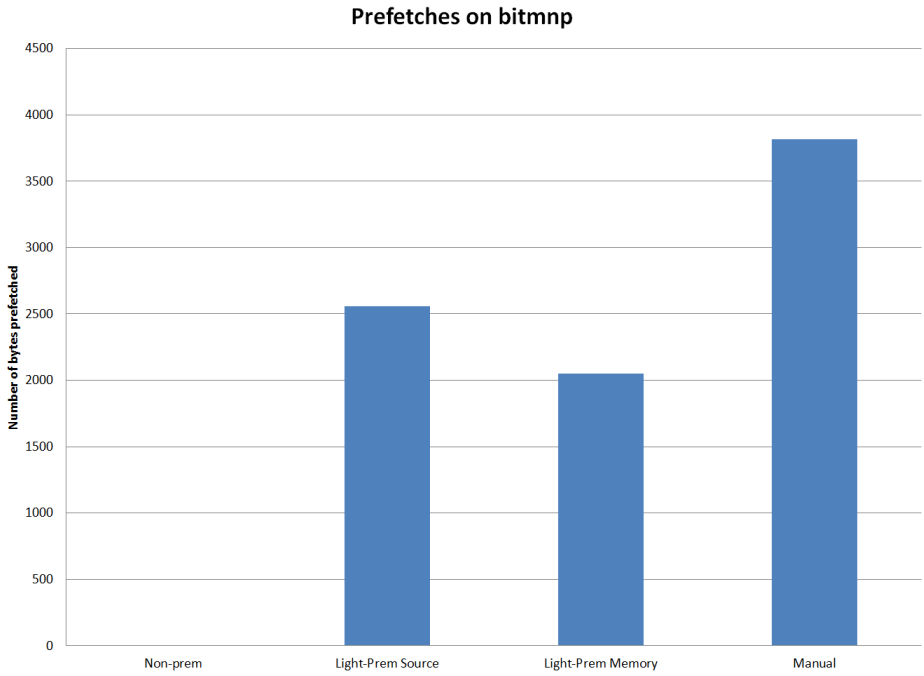
(b) Prefetches

Figure A.2: Cache Misses and Prefetches on basefp

A.1.3 bitmnp



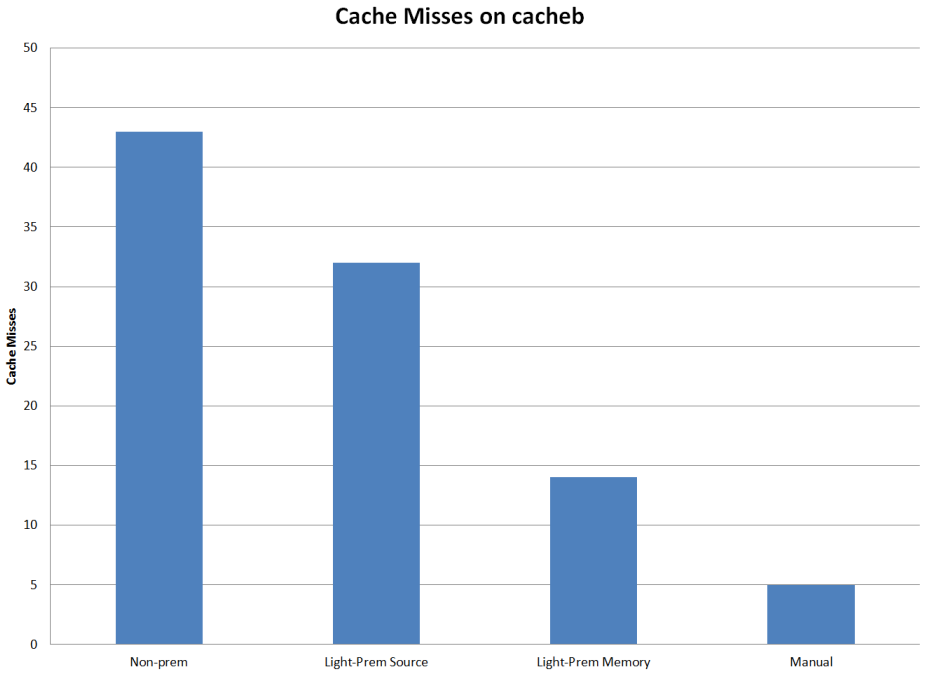
(a) Cache Misses



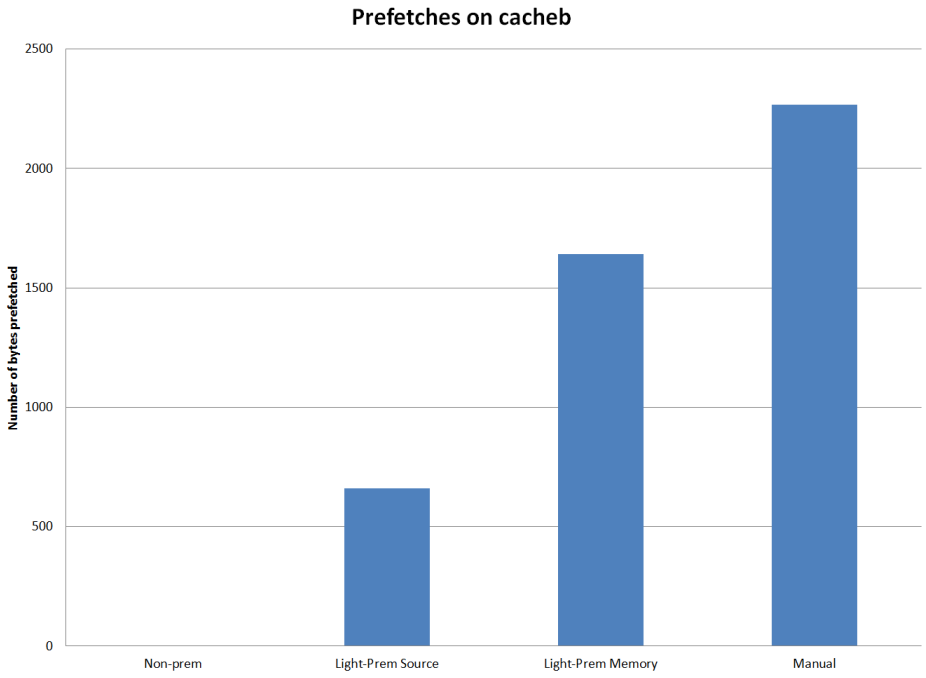
(b) Prefetches

Figure A.3: Cache Misses and Prefetches on bitmnp

A.1.4 cacheb



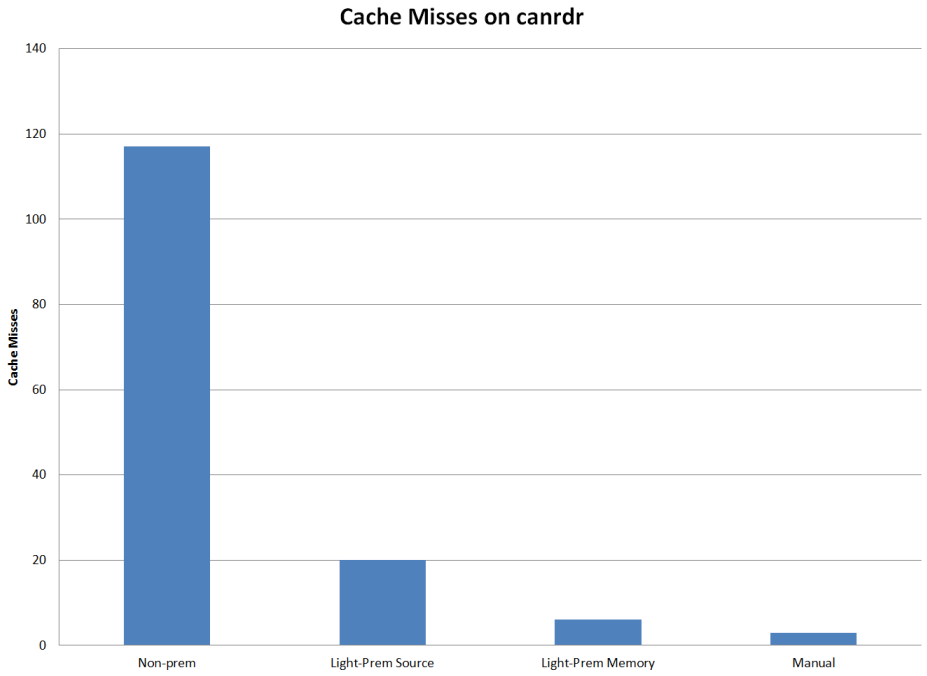
(a) Cache Misses



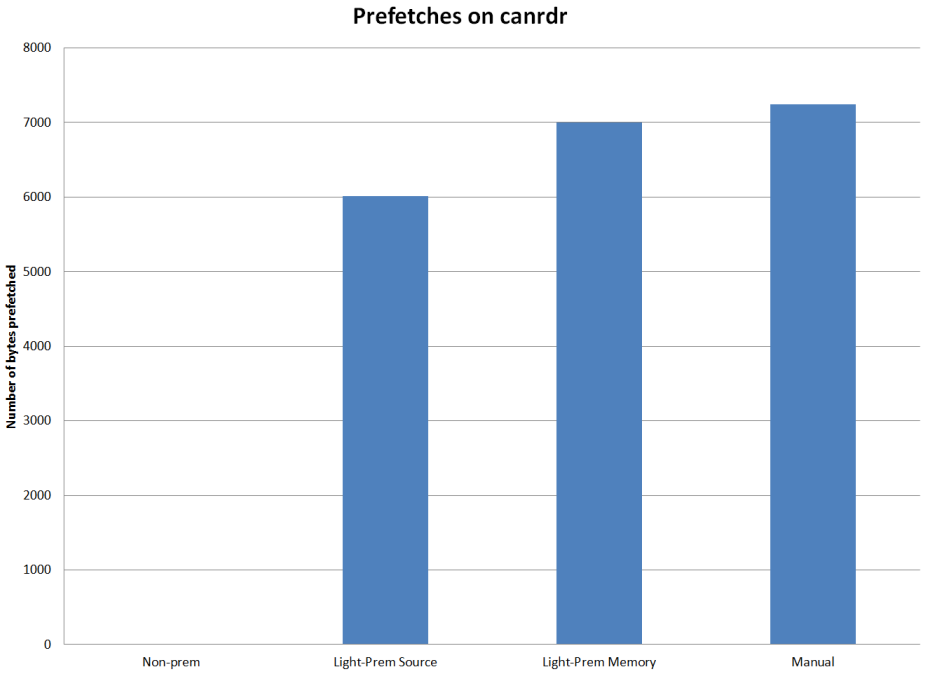
(b) Prefetches

Figure A.4: Cache Misses and Prefetches on cacheb

A.1.5 canrdr



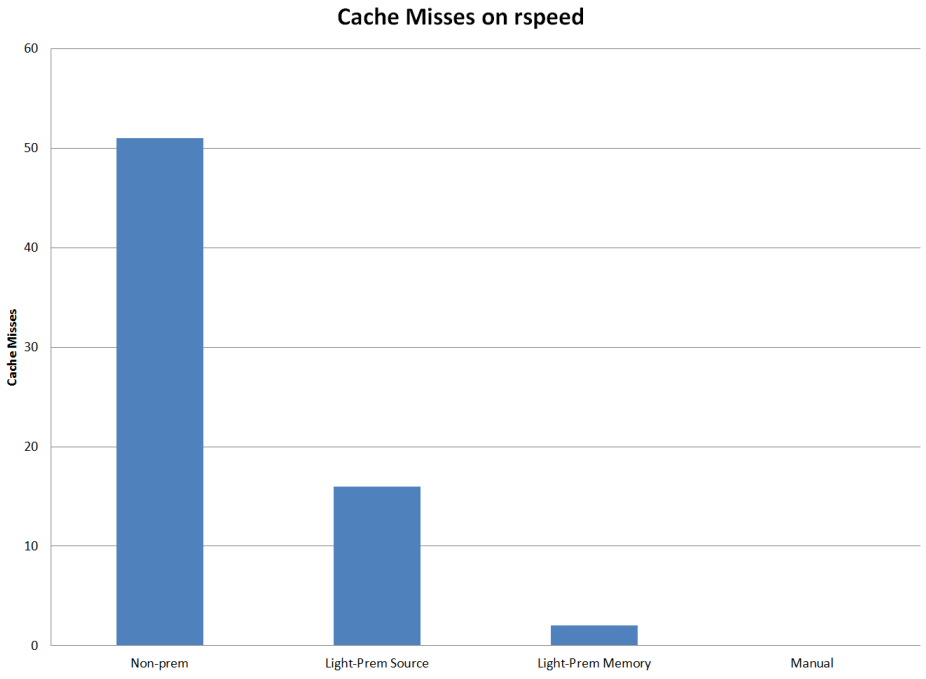
(a) Cache Misses



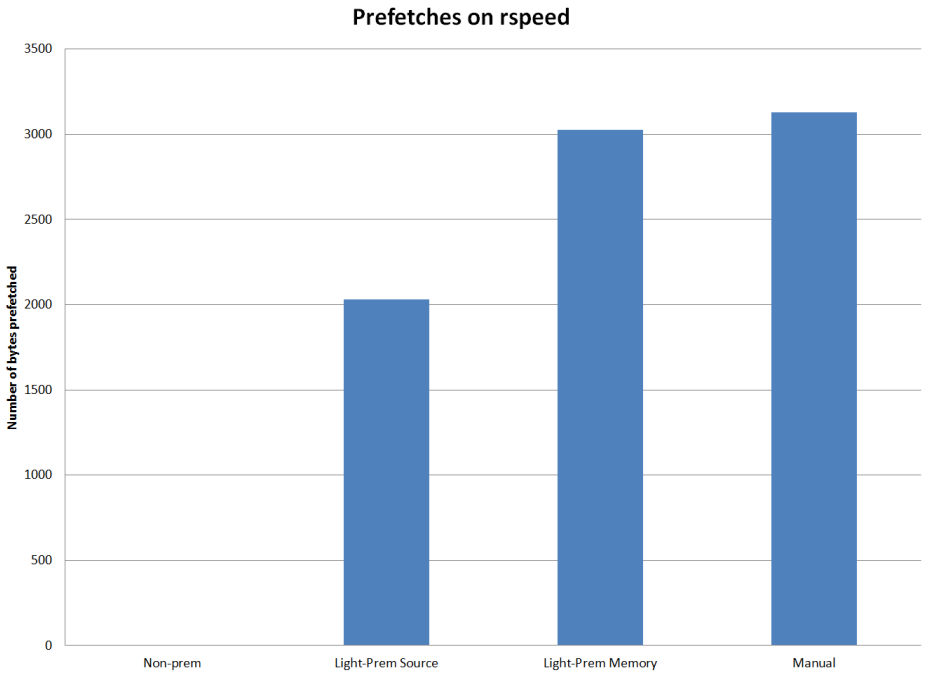
(b) Prefetches

Figure A.5: Cache Misses and Prefetches on canrdr

A.1.6 rspeed



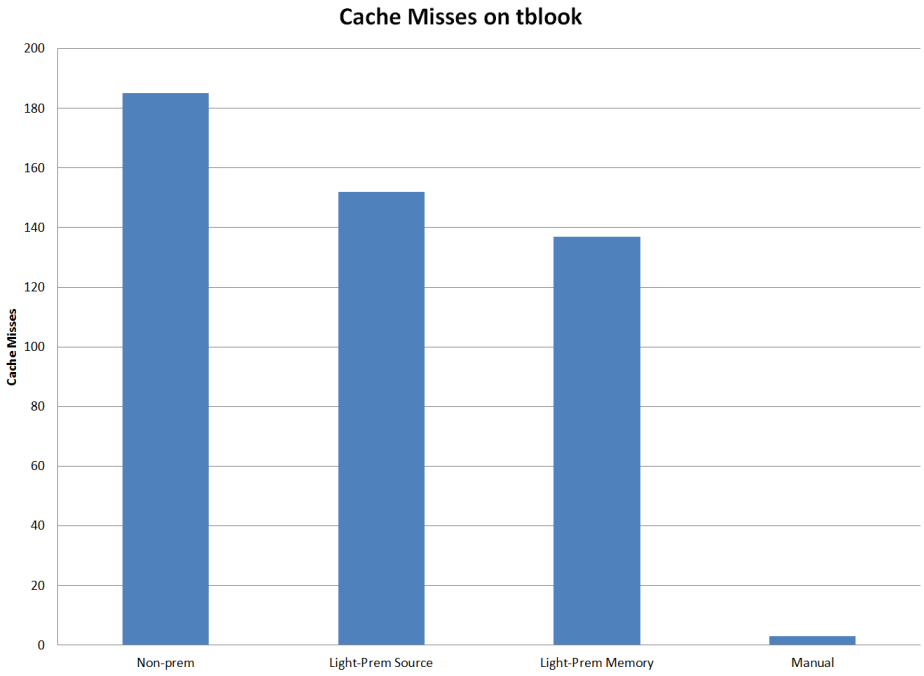
(a) Cache Misses



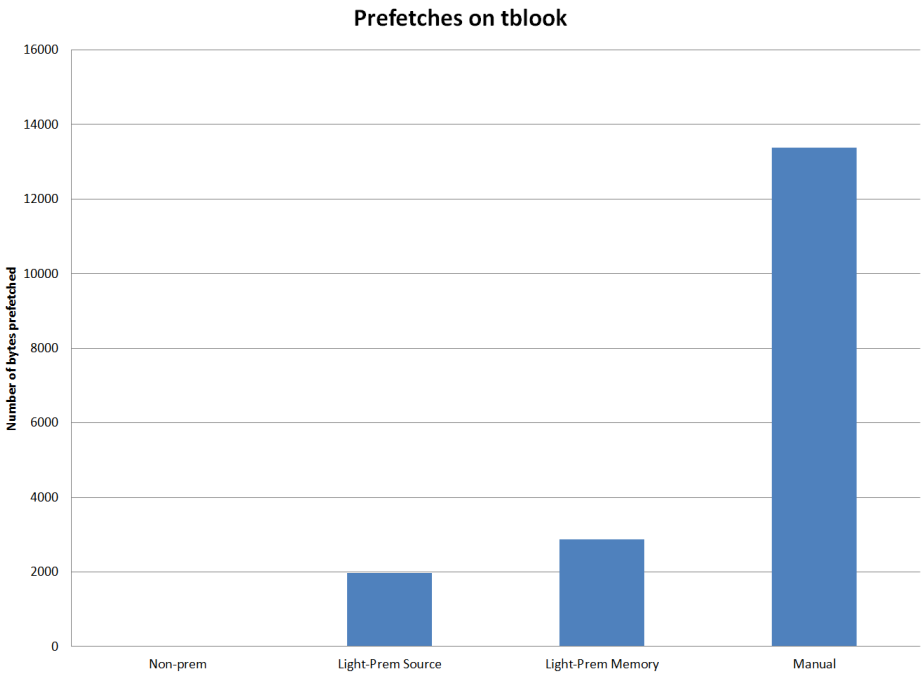
(b) Prefetches

Figure A.6: Cache Misses and Prefetches on rspeed

A.1.7 tblook



(a) Cache Misses



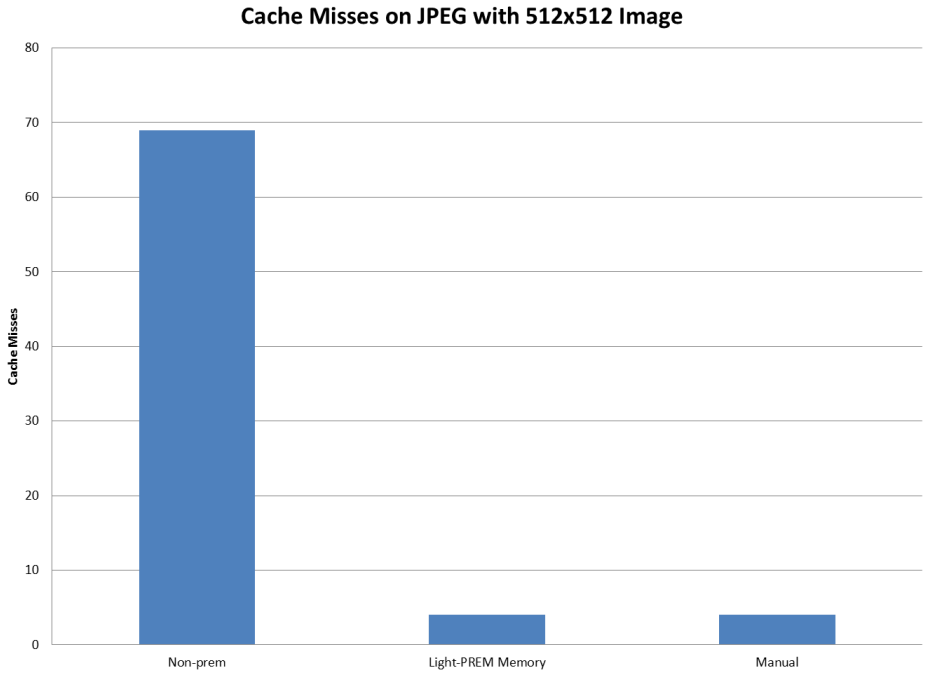
(b) Prefetches

Figure A.7: Cache Misses and Prefetches on tblook

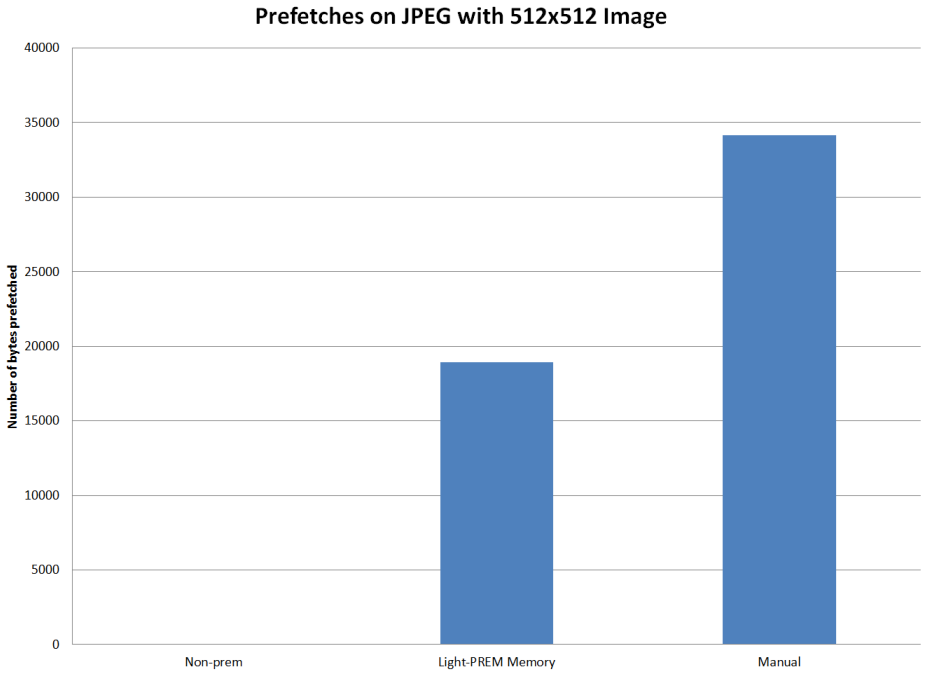
A.2 JPEG Experimental Results

The following figures show the number of cache misses and the number of prefetched bytes on the JPEG benchark after having generated code using Light-PREM. Note that because the Light-PREM Source Analyzer was ineffective, it is not displayed. Instead, only results using the Light-PREM Memory Analyzer are shown.

A.2.1 512x512 image



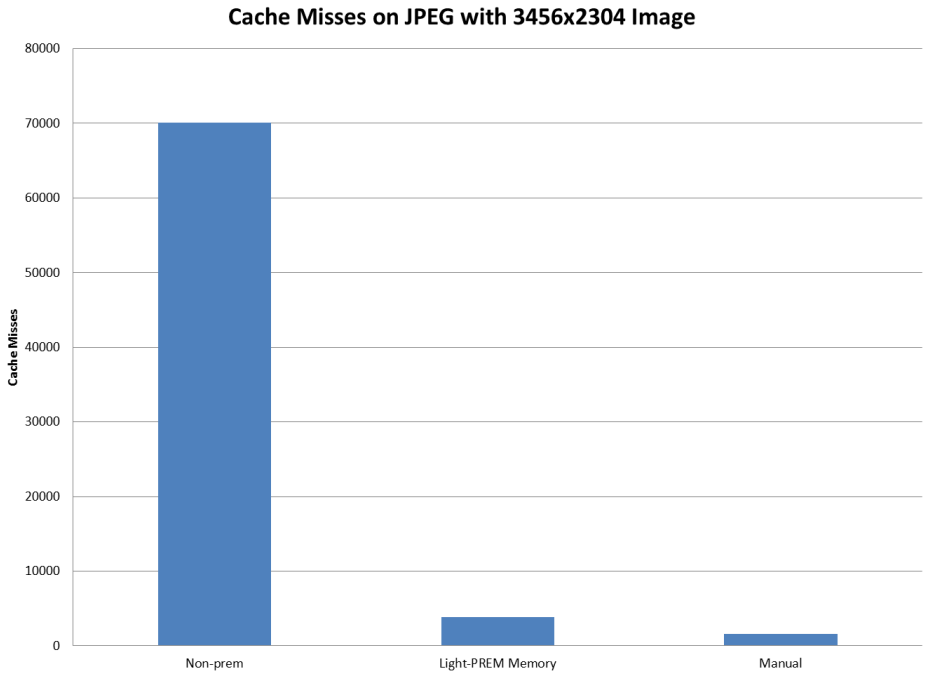
(a) Cache Misses



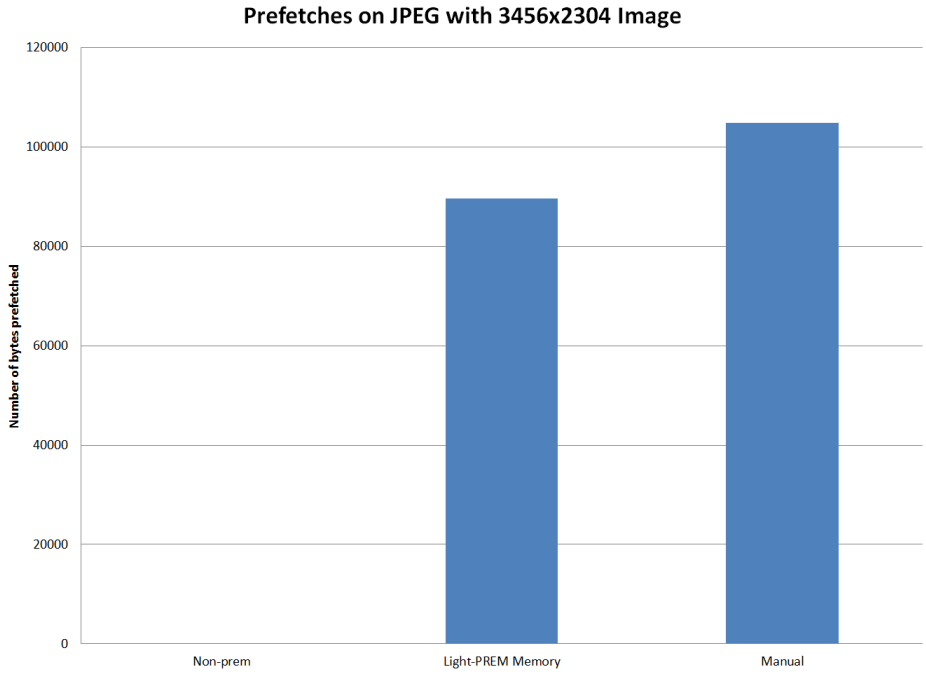
(b) Prefetches

Figure A.8: Cache Misses and Prefetches on JPEG with 512x512 image

A.2.2 3456x2304 image



(a) Cache Misses



(b) Prefetches

Figure A.9: Cache Misses and Prefetches on JPEG with 3456x2304 image

Appendix B

Code Reference

This appendix lists some code samples of Light-PREM generated code, and the majority of the source code of both versions of Light-PREM.

B.1 Sample Light-PREM Generated Code

This section lists PREM prefetch code that was generated by Light-PREM.

B.1.1 Light-PREM Source Analyzer on tblook

Code generated by Light-PREM Source Analyzer on the EEMBC benchmark tblook.

```
n_void predict_tblook(size_t iterations)
{
        PREM_BEGIN_PRED_INTERVAL(0);

        PREM_PREFETCH_GLOBAL( &RAMfilePtr , sizeof(RAMfilePtr) );
        PREM_PREFETCH_GLOBAL( &angleTable , sizeof(angleTable) );
        PREM_PREFETCH_GLOBAL( &engLoad , sizeof(engLoad) );
        PREM_PREFETCH_GLOBAL( &engLoadDelta1 , sizeof(engLoadDelta1) );
        PREM_PREFETCH_GLOBAL( &engLoadDelta2 , sizeof(engLoadDelta2) );
        PREM_PREFETCH_GLOBAL( &engLoadDelta3 , sizeof(engLoadDelta3) );
        PREM_PREFETCH_GLOBAL( &engSpeed , sizeof(engSpeed) );
        PREM_PREFETCH_GLOBAL( &engSpeedDelta1 , sizeof(engSpeedDelta1) );
        PREM_PREFETCH_GLOBAL( &engSpeedDelta2 , sizeof(engSpeedDelta2) );
        PREM_PREFETCH_GLOBAL( &engSpeedDelta3 , sizeof(engSpeedDelta3) );
        PREM_PREFETCH_GLOBAL( &i1 , sizeof(i1) );
        PREM_PREFETCH_GLOBAL( &i2 , sizeof(i2) );
        PREM_PREFETCH_GLOBAL( &i3 , sizeof(i3) );
        PREM_PREFETCH_GLOBAL( inpLoadValue , ( NUM_TESTS + 1 ) * sizeof( varsize ) );
        PREM_PREFETCH_GLOBAL( inpSpeedValue , ( NUM_TESTS + 1 ) * sizeof( varsize ) );
        PREM_PREFETCH_GLOBAL( &isTableLooped , sizeof(isTableLooped) );
        PREM_PREFETCH_GLOBAL( &iterations , sizeof(iterations) );
        PREM_PREFETCH_GLOBAL( &j2 , sizeof(j2) );
        PREM_PREFETCH_GLOBAL( &j3 , sizeof(j3) );
        PREM_PREFETCH_GLOBAL( &j1 , sizeof(j1) );
        PREM_PREFETCH_GLOBAL( &loadValue , sizeof(loadValue) );
        PREM_PREFETCH_GLOBAL( &loop_cnt , sizeof(loop_cnt) );

        PREM_PREFETCH_GLOBAL( &numXEntries , sizeof(numXEntries) );
        PREM_PREFETCH_GLOBAL( &outAngleValue1 , sizeof(outAngleValue1) );
        PREM_PREFETCH_GLOBAL( &outAngleValue2 , sizeof(outAngleValue2) );
        PREM_PREFETCH_GLOBAL( &outAngleValue3 , sizeof(outAngleValue3) );
        PREM_PREFETCH_GLOBAL( &speedValue , sizeof(speedValue) );
        PREM_PREFETCH_GLOBAL( &tableCount , sizeof(tableCount) );


        PREM_BEGIN_PRED_EXECUTION("predict_tblook",0);

        ...
        // rest of code
        ...

        PREM_END_PRED_INTERVAL("predict_tblook",0,0);
}
```

B.1.2 Light-PREM Memory Analyzer on tblook

Code generated by Light-PREM Memory Analyzer on the EEMBC benchmark tblook.

```
n_void predict_tblook(size_t iterations)
{
        PREM_BEGIN_PRED_INTERVAL(0); \
        PREM_PREFETCH_GLOBAL( ((((void*) RAMfile) - 0) + 0), 1024); \
        PREM_PREFETCH_GLOBAL( ((((void*) inpLoadValue) - 0) + 0), 928); \
        PREM_PREFETCH_GLOBAL( ((((void*) inpSpeedValue) - 0) + 0), 928); \
        PREM_PREFETCH_STACK(sizeof(size_t), 0x0); \
        PREM_BEGIN_PRED_EXECUTION("predict_tblook",0);

        ...
        // rest of code
        ...

        PREM_END_PRED_INTERVAL("predict_tblook",0,0);
}
```

B.1.3 Manual on tblook

Code generated by humans after examining and understanding the source code of the EEMBC benchmark tblook.

```
n_void predict_tblook(size_t iterations)
{

  PREM_BEGIN_PRED_INTERVAL(0);

#ifdef PREM_PATCH
  /* add all the prefetch here  */
  PREM_PREFETCH_GLOBAL(RAMfile,  RAMfileSize * sizeof(n_int) + sizeof (varsize) ) ;
  PREM_PREFETCH_GLOBAL(engSpeed,  sizeof (engSpeedROM));
  PREM_PREFETCH_GLOBAL(engLoad,   sizeof (engLoadROM ) );
  PREM_PREFETCH_GLOBAL(angleTable, sizeof(angleTableROM) ) ;
  PREM_PREFETCH_GLOBAL(inpLoadValue,   ( NUM_TESTS + 1 ) * sizeof( varsize ) ) ;
  PREM_PREFETCH_GLOBAL(inpSpeedValue,   ( NUM_TESTS + 1 ) * sizeof( varsize ) ) ;

  PREM_PREFETCH_GLOBAL(&isTableLooped ,   sizeof (n_int) ) ;
  PREM_PREFETCH_GLOBAL(&tableCount ,   sizeof (n_int) ) ;
  PREM_PREFETCH_GLOBAL(&loop_cnt ,   sizeof (n_int) ) ;
  PREM_PREFETCH_GLOBAL(&loadValue,   sizeof (varsize) ) ;
  PREM_PREFETCH_GLOBAL(&speedValue,   sizeof (varsize) ) ;
  PREM_PREFETCH_GLOBAL(&i1 ,   sizeof (varsize ) ) ;
  PREM_PREFETCH_GLOBAL(&i2 ,   sizeof (varsize ) ) ;
  PREM_PREFETCH_GLOBAL(&i3 ,   sizeof (varsize ) ) ;
  PREM_PREFETCH_GLOBAL(&j1 ,   sizeof (varsize ) ) ;
  PREM_PREFETCH_GLOBAL(&j2 ,   sizeof (varsize ) ) ;
  PREM_PREFETCH_GLOBAL(&j3 ,   sizeof (varsize ) ) ;
  PREM_PREFETCH_GLOBAL(&numXEntries,   sizeof (varsize ) ) ;
  PREM_PREFETCH_GLOBAL(&numYEntries,   sizeof (varsize ) ) ;
  PREM_PREFETCH_GLOBAL(&isTableLooped,   sizeof (n_int) ) ;
  PREM_PREFETCH_GLOBAL(&outAngleValue1 ,   sizeof (varsize ) ) ;
  PREM_PREFETCH_GLOBAL(&outAngleValue2,   sizeof (varsize ) ) ;
  PREM_PREFETCH_GLOBAL(&outAngleValue3 ,   sizeof (varsize ) ) ;
  PREM_PREFETCH_GLOBAL(&engSpeedDelta1 ,   sizeof (n_float) ) ;
  PREM_PREFETCH_GLOBAL(&engSpeedDelta2 ,   sizeof (n_float) ) ;
  PREM_PREFETCH_GLOBAL(&engSpeedDelta3 ,   sizeof (n_float) ) ;
  PREM_PREFETCH_GLOBAL(&engLoadDelta1 ,   sizeof (n_float) ) ;
  PREM_PREFETCH_GLOBAL(&engLoadDelta2 ,   sizeof (n_float) ) ;
  PREM_PREFETCH_GLOBAL(&engLoadDelta3 ,   sizeof (n_float) ) ;

  PREM_PREFETCH_STACK(sizeof(size_t), 0x0);
  PREM_BEGIN_PRED_EXECUTION("tblook",0);

#endif  /* end prefetch phase */

        ...
        // rest of code
        ...

        PREM_END_PRED_INTERVAL("tblook",0);
}
```

B.1.4 Light-PREM Memory Analyzer on JPEG

Code generated by Light-PREM Memory Analyzer on a 3456x2304 image:

```
/*
 * Process some data in the single-pass case.
 * We process the equivalent of one fully interleaved MCU row ("iMCU" row)
 * per call, ie, v_samp_factor block rows for each component in the image.
 * Returns TRUE if the iMCU row is completed, FALSE if suspended.
 *
 * NB: input_buf contains a plane for each component in image.
 * For single pass, this is the same as the components in the scan.
 */
METHODDEF(boolean)
compress_data2 (j_compress_ptr cinfo, JSAMPIMAGE input_buf)
{
        PREM_BEGIN_PRED_INTERVAL(0); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) cinfo + 60)) + 16), 12); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) cinfo + 60)) + 52), 8); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) cinfo + 60)) + 64), 12); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) cinfo + 244)) + 16), 12); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) cinfo + 244)) + 52), 8); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) cinfo + 244)) + 64), 12); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) cinfo + 248)) + 16), 12); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) cinfo + 248)) + 52), 8); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) cinfo + 248)) + 64), 12); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) cinfo + 20)) + 0), 8); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) cinfo + 20)) + 12), 4); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) cinfo + 20)) + 20), 8); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 20) + 20)) + 0), 36); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 20) + 20)) + 56), 8); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 20) + 20)) + 68), 16); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 20) + 20)) + 104), 4); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 20) + 20)) + 148), 4); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)(*(void**)((void*) cinfo + 20) + 20) + 72)) + 0), 12);\
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 20) + 24)) + 0), 626); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 20) + 0)) + 0), 3476); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) cinfo + 352)) + 4), 16); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) cinfo + 356)) + 4), 4); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) cinfo + 356)) + 12), 24); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) cinfo + 356)) + 44), 8); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) cinfo + 356)) + 60), 8); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) cinfo + 336)) + 8), 40); \
        PREM_PREFETCH_GLOBAL( (((void*) input_buf) + 0), 12); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) input_buf + 0)) + 0), 64); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) input_buf + 4)) + 0), 32); \
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) input_buf + 8)) + 0), 32); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 352) + 12)) + 0), 256); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 352) + 16)) + 0), 256); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 336) + 24)) + 0), 130); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 336) + 28)) + 0), 130); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 336) + 32)) + 0), 130); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 336) + 36)) + 0), 130); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 336) + 40)) + 0), 130); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 336) + 44)) + 0), 130); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 0) + 0)) + 0), 3459); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 0) + 4)) + 0), 3459); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 0) + 8)) + 0), 3459); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 0) + 12)) + 0), 3459); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 0) + 16)) + 0), 3459); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 0) + 20)) + 0), 3459); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 0) + 24)) + 0), 3459); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 0) + 28)) + 0), 3459); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 0) + 32)) + 0), 3459); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 0) + 36)) + 0), 3459); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 0) + 40)) + 0), 3459); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 0) + 44)) + 0), 3459); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 0) + 48)) + 0), 3459); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 0) + 52)) + 0), 3459); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 0) + 56)) + 0), 3459); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 0) + 60)) + 0), 3459); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 4) + 0)) + 0), 1731); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 4) + 4)) + 0), 1731); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 4) + 8)) + 0), 1731); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 4) + 12)) + 0), 1731); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 4) + 16)) + 0), 1731); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 4) + 20)) + 0), 1731); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 4) + 24)) + 0), 1731); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 4) + 28)) + 0), 1731); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 8) + 0)) + 0), 1731); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 8) + 4)) + 0), 1731); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 8) + 8)) + 0), 1731); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 8) + 12)) + 0), 1731); \
        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 8) + 16)) + 0), 1731); \
```

```
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 8) + 20)) + 0), 1731); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 8) + 24)) + 0), 1731); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) input_buf + 8) + 28)) + 0), 1731); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 44)) + 0), 36); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 44)) + 1024), 12); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 0), 36); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 68), 28); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 132), 28); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 196), 28); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 260), 24); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 324), 20); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 388), 16); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 452), 16); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 516), 16); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 580), 20); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 644), 16); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 708), 12); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 772), 12); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 836), 12); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 900), 12); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 960), 12); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 1024), 12); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 1041), 10); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 1057), 10); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 1073), 10); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 1089), 9); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 1105), 8); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 1121), 7); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 1137), 7); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 1153), 7); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 1169), 8); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 1185), 7); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 1201), 6); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 1217), 6); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 1233), 6); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 1249), 6); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 60)) + 1264), 6); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 48)) + 0), 28); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 48)) + 1024), 10); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 0), 24); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 68), 20); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 132), 16); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 196), 12); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 260), 12); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 324), 8); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 388), 4); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 452), 4); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 516), 4); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 580), 4); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 644), 4); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 708), 4); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 772), 4); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 836), 4); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 900), 4); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 960), 8); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 1024), 9); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 1041), 8); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 1057), 7); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 1073), 6); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 1089), 6); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 1105), 5); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 1121), 4); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 1137), 4); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 1153), 4); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 1169), 4); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 1185), 4); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 1201), 4); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 1217), 4); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 1233), 4); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 1249), 4); \
PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) cinfo + 356) + 64)) + 1264), 5); \
PREM_PREFETCH_GLOBAL( (((void*) cinfo) + 20), 4); \
PREM_PREFETCH_GLOBAL( (((void*) cinfo) + 192), 4); \
PREM_PREFETCH_GLOBAL( (((void*) cinfo) + 232), 20); \
PREM_PREFETCH_GLOBAL( (((void*) cinfo) + 256), 4); \
PREM_PREFETCH_GLOBAL( (((void*) cinfo) + 264), 28); \
PREM_PREFETCH_GLOBAL( (((void*) cinfo) + 336), 4); \
PREM_PREFETCH_GLOBAL( (((void*) cinfo) + 352), 8); \
PREM_PREFETCH_STACK(sizeof(size_t), 0x0); \
PREM_BEGIN_PRED_EXECUTION("compress_data2",0);

...
// rest of code
...
```

```
        PREM_END_PRED_INTERVAL("compress_data2",0);

        return TRUE;
}
```

B.1.5 Manual on JPEG

Code generated by humans after examining and understanding the JPEG source code.

```
/*
 * Process some data in the single-pass case.
 * We process the equivalent of one fully interleaved MCU row ("iMCU" row)
 * per call, ie, v_samp_factor block rows for each component in the image.
 * Returns TRUE if the iMCU row is completed, FALSE if suspended.
 *
 * NB: input_buf contains a plane for each component in image.
 * For single pass, this is the same as the components in the scan.
 */

METHODDEF(boolean)
predict_compress_data (j_compress_ptr cinfo, JSAMPIMAGE input_buf)
{
  PREM_BEGIN_PRED_INTERVAL(0);

  my_coef_ptr coef = (my_coef_ptr) cinfo->coef;
  JDIMENSION MCU_col_num;       /* index of current MCU within row */
  //JDIMENSION last_MCU_col = cinfo->MCUs_per_row - 1;
  //JDIMENSION last_iMCU_row = cinfo->total_iMCU_rows - 1;
  JDIMENSION last_MCU_col;
  JDIMENSION last_iMCU_row;
  int blkn, bi, ci, yindex, yoffset, blockcnt;
  JDIMENSION ypos, xpos;
  jpeg_component_info *compptr;

#if 1  //comment the data prefetch.  check the reason of large number of prefetch.
  unsigned input_buf_addr = input_buf[0][0];
  unsigned MCU_buf_addr = coef->MCU_buffer[0];
  unsigned comp_0_addr = cinfo->cur_comp_info[0];
  unsigned comp_1_addr = cinfo->cur_comp_info[1];
  unsigned comp_2_addr = cinfo->cur_comp_info[2];
  my_fdct_ptr fdct_t = (my_fdct_ptr)cinfo->fdct;
  unsigned fdct_divisor_addr = fdct_t->divisors[0];
  huff_entropy_ptr entropy_t = (huff_entropy_ptr) cinfo->entropy;
  my_destination_mgr *dest_addr = cinfo->dest;
  char *buff = dest_addr->buffer;
  unsigned next_output_addr = cinfo->dest->next_output_byte;
  unsigned entropy_dc_tbl_addr =((huff_entropy_ptr) cinfo->entropy)->dc_derived_tbls[0];
  unsigned entropy_ac_tbl_addr =((huff_entropy_ptr) cinfo->entropy)->ac_derived_tbls[0];
  unsigned count_addr  = & count;

  PREM_PREFETCH_GLOBAL(cinfo,sizeof(struct jpeg_compress_struct));
  PREM_PREFETCH_GLOBAL(input_buf_addr, (cinfo->image_width*24+32));   //16+4+4
 /*be conservative, can use 6 instead of C_MAX_BLOCK_IN_MCU */
  PREM_PREFETCH_GLOBAL(MCU_buf_addr, sizeof(JBLOCK)*C_MAX_BLOCKS_IN_MCU);
  PREM_PREFETCH_GLOBAL(comp_0_addr, sizeof(jpeg_component_info));
  PREM_PREFETCH_GLOBAL(comp_1_addr, sizeof(jpeg_component_info));
  PREM_PREFETCH_GLOBAL(comp_2_addr, sizeof(jpeg_component_info));
  PREM_PREFETCH_GLOBAL(fdct_t, sizeof(my_fdct_controller));
  PREM_PREFETCH_GLOBAL(fdct_divisor_addr, sizeof(DCTELEM)*DCTSIZE2* NUM_QUANT_TBLS);
  PREM_PREFETCH_GLOBAL(entropy_t, sizeof(huff_entropy_encoder));
  PREM_PREFETCH_GLOBAL(dest_addr, sizeof(my_destination_mgr));
  PREM_PREFETCH_GLOBAL(buff, 4096);
  PREM_PREFETCH_GLOBAL(next_output_addr, 4096 ); /* OUTPUT_BUF_SIZE 4096 defined in jdatadst.c */
  PREM_PREFETCH_GLOBAL(entropy_dc_tbl_addr, sizeof(c_derived_tbl)*NUM_HUFF_TBLS );
  PREM_PREFETCH_GLOBAL(entropy_ac_tbl_addr,  sizeof(c_derived_tbl)*NUM_HUFF_TBLS );
  PREM_PREFETCH_GLOBAL(jpeg_natural_order, (sizeof(int)*(DCTSIZE2+16)));
  PREM_PREFETCH_GLOBAL(count_addr, (sizeof(struct PERF_COUNTERS))  );
  PREM_PREFETCH_STACK((sizeof(j_compress_ptr)+sizeof(JSAMPIMAGE)),0x0)
#endif

  PREM_BEGIN_PRED_INTERVAL("predict_compress_data", 0);

  ...
  // rest of code
  ...

  PREM_END_PRED_INTERVAL("predict_compress_data", 0);

  return TRUE:
}
```

B.1.6 Light-PREM Memory Analyzer on Simple Example

The following code shows a simple and complete sample program, and the prefetch statements generated by Light-PREM Memory Analyzer

```
#include <stdio.h>
#include <stdlib.h>
#include "lightprem.h"

#define SIZE (1024*4*1)

typedef struct struct1 {
        double dummy1, dummy2;
        int *ptr;
} struct1;

typedef struct struct2 {
        double dummy1, dummy2, dummy3;
        char dummy4;
        struct1 *ptr;
} struct2;

void foo(struct2 *struct2_ptr) {
        PREM_BEGIN_PRED_INTERVAL(0);

        PREM_PREFETCH_GLOBAL( ((*(void**)(*(void**)((void*) struct2_ptr + 28) + 16)) + 0), 16384);
        PREM_PREFETCH_GLOBAL( ((*(void**)((void*) struct2_ptr + 28)) + 16), 4);
        PREM_PREFETCH_GLOBAL( (((void*) struct2_ptr) + 28), 4);

        PREM_PREFETCH_STACK(sizeof(size_t), 0x0);
        PREM_BEGIN_PRED_EXECUTION("foo",0);

        int i, *int_ptr = struct2_ptr->ptr->ptr;

        for (i = 0; i < SIZE; i++)
                int_ptr[i] = 2*i; // some dummy calculation

        PREM_END_PRED_INTERVAL("foo",0,0);
}

int main() {
        int i, *array = (int *) malloc(SIZE * sizeof(int));

        for (i = 0; i < SIZE; i++)
                array[i] = i;

        // allocate structures
        struct1 *struct1_ptr = (struct1 *) malloc(sizeof(struct1));
        struct2 *struct2_ptr = (struct2 *) malloc(sizeof(struct2));

        // set up so that struct2_ptr->ptr->ptr == array
        struct2_ptr->ptr = struct1_ptr;
        struct1_ptr->ptr = array;

        // call "predictable interval"
        foo(struct2_ptr);

        return 0;
}
```

B.2 Light-PREM Source Analyzer Code

This section lists configuration files and source code of the Light-PREM Source Analyzer.

B.2.1 Sample Configuration File

The following is an example configuration file that lists all possble options for Light-PREM Source Analyzer. A configuration file like this one must be created in order to run Light-PREM Source Analyzer.

```
source=/path/to/your/source/code
```

```
function=foo
executable=foo.exe
executable_options=-help
preprocessor_opts=-I ./
makefile=makefile.alt
callgrind_out="callgrind.out"
temp_file="file.tmp"
```

B.2.2 Light-PREM Source Analyzer Script

The following bash script runs Light-PREM Source Analyzer. Note that the script calls other executables, some of which are included in this appendix as well, and some of which are not included.

```
#!/bin/bash


#############################################################
####                  helper functions                   ####
#############################################################

usage(){
        echo "Usage: light_prem <config file>"
        exit 1
}

check(){
        if [ ! -e $1 ]; then
                echo "Error: \"$1\" does not exist"
                exit 1
        elif [ ! -r $1 ]; then
                echo "Error: insufficient privileges to read \"$1\""
                exit 1
        fi
}

exists() {
        if [ -z "${!1}" ]; then
                echo "Error: \"$1\" must be set in the config file"
                exit 1
        fi
}


#############################################################
####                       setup                         ####
#############################################################

TOOLDIR=/home/rtsl/svn/predictableCOTS/software/light-prem/tool
FORMAT_CFG=$TOOLDIR/config/format.cfg

if [ $# -lt 1 ]; then
        usage
fi

check $TOOLDIR/config/default.cfg
source $TOOLDIR/config/default.cfg

CONFIG=$1

check $CONFIG
source $CONFIG

exists "source"
check $source
exists "function"
exists "executable"
exists "callgrind_out"
exists "malloc_file"
exists "array_file"

if [ -z "$temp_file" ]; then
        temp_file=$source.tmp
fi

#############################################################
```

```
####                    format files                        ####
################################################################

echo "running preprocessor/formatter..."
if ! gcc $preprocessor_opts -E $source &> /dev/null; then
        echo "Error: could not run preprocessor on \"$source\""
        gcc $preprocessor_opts -E $source > /dev/null
        exit 1
fi

cp $source $source.orig
gcc $preprocessor_opts -E $source | grep -v ^# | tr '\n' ' ' | \
sed 's/  */ /g' | uncrustify -c $FORMAT_CFG > $temp_file

check $temp_file
echo >> $temp_file

cp $temp_file $source
rm $temp_file


################################################################
####                  create executable                     ####
################################################################

if [ -z "$makefile" ]; then
        make $executable
else
        make --makefile=$makefile $executable
fi

check $executable


################################################################
####                    run callgrind                       ####
################################################################

echo "running callgrind on executable \"$executable\"..."
valgrind --tool=callgrind --cache-sim=yes --simulate-wb=yes --I1=32768,8,64 \
--D1=32768,8,64 --LL=131072,16,64 --callgrind-out-file=$callgrind_out \
$executable $executable_options &> /dev/null
check $callgrind_out


################################################################
####                 find mallocs/arrays                    ####
################################################################

echo "finding malloc calls and arrays..."
check $TOOLDIR/bin/cg_format
check $TOOLDIR/bin/prefetch
$TOOLDIR/bin/cg_format $callgrind_out | $TOOLDIR/bin/prefetch malloc > $malloc_file
check $malloc_file
$TOOLDIR/bin/cg_format $callgrind_out | $TOOLDIR/bin/prefetch array > $array_file
check $array_file
rm $callgrind_out


################################################################
####                    run callgrind                       ####
################################################################

echo "running callgrind for function \"$function\"..."
valgrind --tool=callgrind --cache-sim=yes --simulate-wb=yes --I1=32768,8,64 \
--D1=32768,8,64 --LL=131072,16,64 --toggle-collect=$function \
--callgrind-out-file=$callgrind_out $executable $executable_options &> /dev/null
check $callgrind_out


################################################################
####                    add prefetches                      ####
################################################################

echo "adding prefetch statements to source file \"$source\"..."
check $TOOLDIR/bin/cg_format
check $TOOLDIR/bin/prefetch
$TOOLDIR/bin/cg_format $callgrind_out | \
$TOOLDIR/bin/prefetch prefetch "$source.orig" "$malloc_file" "$array_file" "$aggresive" > $temp_file
check $temp_file
cp $temp_file $source
rm $temp_file
rm $callgrind_out
rm $malloc_file
rm $array_file
```

```
rm $source.orig


#############################################################
####            remove extraneous prefetchs              ####
#############################################################

if [ "$skip_remove" = "true" ]; then
        exit 0
fi

echo "removing extra prefetches..."
while [ 1 ]; do
        if gcc $preprocessor_opts -c $source &> /dev/null; then
                break
        else
                LINE=$(gcc $preprocessor_opts -c $source 2>&1 | \
                grep  "$source:[0-9][0-9]*" | head -1 | sed "s/$source:\([0-9][0-9]*\).*/\1/")
                sed "$LINE s/.*//" < $source > $temp_file
                cp $temp_file $source
                rm $temp_file
        fi
done
```

B.2.3 Light-PREM Source Analyzer Code

The code to the Light-PREM Source Analyzer is shown below.

```
#include <string>
#include <string.h>
#include <vector>
#include <iostream>
#include <fstream>
#include <sstream>
#include <map>
#include <stdlib.h>
#include <algorithm>
using namespace std;

#include "prefetch.h"
#include "prefetch_malloc.h"
#include "prefetch_adder.h"
#include "prefetch_parser.h"
#include "prefetch_array.h"
#include "regex.h"

void usage() {
        cerr << "usage: " << endl;
        cerr << "\tprefetch malloc" << endl;
        cerr << "\tprefetch array" << endl;
        cerr << "\tprefetch prefetch SOURCE_FILE MALLOC_FILE ARRAY_FILE [aggresive = false]" << endl;
        exit(1);
}

int main(int argc, const char* argv[]) {
        bool run_malloc = false;
        bool run_prefetch = false;
        bool run_array = false;
        bool aggresive = false;

        if (argc < 2)
                usage();

        if (strcmp(argv[1], "prefetch") == 0) {
                if (!(argc == 5 || argc == 6))
                        usage();
                if (argc == 6)
                        aggresive = (strcmp(argv[5], "true") == 0);
                run_prefetch = true;
        }
        else if (strcmp(argv[1], "malloc") == 0) {
                if (argc != 2)
                        usage();
                run_malloc = true;
        }
        else if (strcmp(argv[1], "array") == 0) {
                if (argc != 2)
                        usage();
                run_array = true;
        }
```

```
        if (!run_malloc && !run_prefetch && !run_array)
                usage();

        map<string, source_file> files = parse();

        if (run_prefetch) {
                add_prefetchs(files, argv[3], argv[4], aggresive);
                print_source(argv[2]);
        }
        else if (run_malloc) {
                print_mallocs(files);
        }
        else if (run_array) {
                print_arrays(files);
        }

        return 0;
}


/********** Prefetch Adder **********/

#define VARIABLE_REGEX "([a-zA-Z_][a-zA-Z0-9_]*)[ ]*[^( a-zA-Z0-9_]"

map<string, string> prefetch_map;
map<string, string> malloc_map;
set<string> arrays;

bool filter(string var) {
        if (match(var, "PREM") || match(var, "prem"))
                return false;
        return true;
}

bool should_add_prefetchs(source_line line) {
        if (match(line.source, "PREM_PREFETCH_GLOBAL") ||
                        match(line.source, "PREM_PREFETCH_STACK") ||
                        match(line.source, "PREM_process_setup") ||
                        match(line.source, "PREM_trash_cache") ||
                        match(line.source, "PREM_BEGIN_PRED_INTERVAL") ||
                        match(line.source, "PREM_BEGIN_PRED_EXECUTION") ||
                        match(line.source, "PREM_END_PRED_INTERVAL") ||
                        match(line.source, "__asm__"))
                return false;

        return line.read_miss > 0 || line.write_miss > 0;
}

/*** add prefetch code ***/

void add_prefetch(string name, string size, bool is_array = false) {
        if (prefetch_map.find(name) != prefetch_map.end())
                return;

        if (is_array)
                prefetch_map[name] = "  PREM_PREFETCH_GLOBAL( " + name + " , " + size + " );";
        else
                prefetch_map[name]="    PREM_PREFETCH_GLOBAL( &" + name + " , " + size + " );";
}

void add_prefetch(string name) {
        if (malloc_map.find(name) != malloc_map.end())
                add_prefetch(name, malloc_map[name], true);
        else if (arrays.find(name) != arrays.end())
                add_prefetch(name, "sizeof(" + name + ")", true);
        else
                add_prefetch(name, "sizeof(" + name + ")", false);
}

void add_prefetchs(source_line line) {
        vector<string> vars = extract_all(line.source, VARIABLE_REGEX);
        for (unsigned int i = 0; i < vars.size(); i++)
                if (filter(vars[i]))
                        add_prefetch(vars[i]);
}

void get_mallocs(string file) {
        ifstream contents(file.c_str());

        if (!contents.is_open()) {
                cerr << "Error: Cannot open file \"" << file << "\"" << endl;
                return;
        }

        vector<string> lines = get_lines(contents);
```

```
        for (unsigned int i = 1; i < lines.size(); i+=2) {
                string name = clean(trim(lines[i-1]));
                string size = clean(lines[i]);

                malloc_map[name] = size;
        }
}

void get_arrays(string file) {
        ifstream contents(file.c_str());

        if (!contents.is_open()) {
                cerr << "Error: Cannot open file \"" << file << "\"" << endl;
                return;
        }

        vector<string> lines = get_lines(contents);
        for (unsigned int i = 0; i < lines.size(); i++) {
                string name = clean(trim(lines[i]));
                arrays.insert(name);
        }
}

void add_prefetchs(map<string, source_file> files, string malloc_file, string array_file, bool aggresive){
        get_mallocs(malloc_file);
        get_arrays(array_file);

        if (aggresive) {
                map<string, string>::iterator mit;
                for (mit = malloc_map.begin(); mit != malloc_map.end(); mit++)
                        add_prefetch(mit->first);

                set<string>::iterator sit;
                for (sit = arrays.begin(); sit != arrays.end(); sit++)
                        add_prefetch(*sit);
        }

        map<string, source_file>::iterator mit;
        for (mit = files.begin(); mit != files.end(); mit++) {
                source_file file = mit->second;
                vector<source_line>::iterator vit;
                for (vit = file.lines.begin(); vit < file.lines.end(); vit++) {
                        if (should_add_prefetchs(*vit))
                                add_prefetchs(*vit);
                }
        }
}

/*** print prefetch code ***/

void print_prefetch() {
        map<string,string>::iterator it;
        for (it = prefetch_map.begin(); it != prefetch_map.end(); it++)
                cout << it->second << endl;
}

void print_source(string source) {
        ifstream contents(source.c_str());

        if (!contents.is_open()) {
                cerr << "Error: Cannot open file \"" << source << "\"" << endl;
                return;
        }

        vector<string> lines = get_lines(contents);
        bool added_prefetch = false;
        for (unsigned int i = 0; i < lines.size(); i++) {
                string line = lines[i];

                if (!added_prefetch && match(line, "PREM_BEGIN_PRED_EXECUTION")) {
                        print_prefetch();
                        added_prefetch = true;
                }

                cout << clean(line) << endl;
        }
}


/********** Prefetch Mallocs **********/

bool contains_malloc(source_line line) {
        return line.instructions > 0 && match(line.source, "malloc");
}
```

```
void print_malloc(source_line line) {
        vector<string> results = extract(line.source,  "(.*)=.*th_malloc[^(]*[(](.*),.*,[0-9 ]*[)]");
        if (results.size() >= 3) {
                cout << results[1] << endl;
                cout << results[2] << endl;
                return;
        }

        results = extract(line.source,  "(.*)=.*th_malloc[^(]*[(](.*)[)]");
        if (results.size() >= 3) {
                cout << results[1] << endl;
                cout << results[2] << endl;
                return;
        }

        results = extract(line.source, "(.*)=.*malloc[^(]*[(](.*)[)]");
        if (results.size() >= 3) {
                cout << results[1] << endl;
                cout << results[2] << endl;
                return;
        }
}

void print_mallocs(map<string, source_file> files) {
        map<string, source_file>::iterator mit;
        for (mit = files.begin(); mit != files.end(); mit++) {
                source_file file = mit->second;
                vector<source_line>::iterator vit;
                for (vit = file.lines.begin(); vit < file.lines.end(); vit++) {
                        if (contains_malloc(*vit))
                                print_malloc(*vit);
                }
        }

}


/********** Prefetch Arrays **********/

#define ARRAY_REGEX "([a-zA-Z_][a-zA-Z0-9_]*)[ \t]*[[][]]"
#define ARRAY_REGEX2 "^([a-zA-Z_][a-zA-Z0-9_]*[ \t])*([a-zA-Z_][a-zA-Z0-9_]*)[ \t]*[[].*[]][ \t]*;"

string get_array(source_line line) {
        vector<string> results = extract(line.source, ARRAY_REGEX);
        if (results.size() >= 2) {
                return results[1];
        }

        results = extract(line.source, ARRAY_REGEX2);
        if (results.size() >= 3) {
                return results[2];
        }

        return "";
}

void print_arrays(map<string, source_file> files) {
        set<string> arrays;

        map<string, source_file>::iterator mit;
        for (mit = files.begin(); mit != files.end(); mit++) {
                source_file file = mit->second;
                vector<source_line>::iterator vit;
                for (vit = file.lines.begin(); vit < file.lines.end(); vit++) {
                        string name = get_array(*vit);
                        if (name.size() > 0)
                                arrays.insert(name);
                }
        }

        set<string>::iterator it;
        for (it = arrays.begin(); it != arrays.end(); it++) {
                cout << *it << endl;
        }

}


/********** Prefetch Parser ***********/

map<string, source_file> parse() {
        map<string, source_file> files;
        vector<string> callgrind = get_lines(cin);
        source_file f;
        bool name = false;
```

```
        for (unsigned int i = 1; i < callgrind.size(); i++) {
                string cg = callgrind[i];

                if (cg.empty()) {
                        files[f.name] = f;
                        name = false;
                        continue;
                }

                if (!name) {
                        name = true;
                        f.name = cg;
                        f.lines = vector<source_line>();
                }
                else {
                        vector<string> tokens = extract(cg, "^([0-9]+),([0-9]+),([0-9]+),(.*)$");
                        source_line l;
                        l.instructions = atoi(tokens[1].c_str());
                        l.read_miss = atoi(tokens[2].c_str());
                        l.write_miss = atoi(tokens[3].c_str());
                        l.source = tokens[4];
                        f.lines.push_back(l);
                }
        }

        files[f.name] = f; //add last file
        return files;
}


/********** Regex helpers **********/

#define REGEX_BUFFER 100

bool match(string str, string pattern) {
        regex_t r;

        if (regcomp(&r, pattern.c_str(), REG_EXTENDED)) {
                fprintf(stderr, "regcomp: bad pattern: '%s'\n", pattern.c_str());
                return false;
        }

        bool result = regexec(&r, str.c_str(), 0, NULL, 0) == 0;
        regfree(&r);

        return result;
}

vector<string> extract(string str, string pattern) {
        regex_t r;
        vector<string> results;

        if (regcomp(&r, pattern.c_str(), REG_EXTENDED)) {
                fprintf(stderr, "regcomp: bad pattern: '%s'\n", pattern.c_str());
                return results;
        }

        regmatch_t buffer[REGEX_BUFFER];
        if (regexec(&r, str.c_str(), REGEX_BUFFER, buffer, 0) != 0) {
                regfree(&r);
                return results;
        }

        for (int i = 0; i < REGEX_BUFFER; i++) {
                if (buffer[i].rm_so < 0 || buffer[i].rm_eo < 0)
                        break;

                string result = str.substr(buffer[i].rm_so, buffer[i].rm_eo - buffer[i].rm_so);
                results.push_back(result);
        }

        regfree(&r);
        return results;
}

vector<string> extract_all(string str, string pattern) {
        regex_t r;
        vector<string> results;
        int offset = 0;

        if (regcomp(&r, pattern.c_str(), REG_EXTENDED)) {
                fprintf(stderr, "regcomp: bad pattern: '%s'\n", pattern.c_str());
                return results;
        }
```

```
        regmatch_t buffer[REGEX_BUFFER];
        while (regexec(&r, str.c_str() + offset, REGEX_BUFFER, buffer, 0) == 0) {
                for (int i = 1; i < REGEX_BUFFER; i++) {
                        if (buffer[i].rm_so < 0 || buffer[i].rm_eo < 0)
                                break;

                        string result = str.substr(offset+buffer[i].rm_so,
                                        buffer[i].rm_eo - buffer[i].rm_so);
                        results.push_back(result);
                }
                offset += buffer[0].rm_eo;
        }

        regfree(&r);
        return results;
}


/********** Helpers **********/

string clean(string input) {
        if (input[input.length() - 1] == '\r')
                return input.substr(0, input.length() - 1);
        return input;
}

vector<string> get_lines(istream &stream) {
        vector<string> lines(2);

        string lineInput;
        while (getline(stream, lineInput)) {
                lines.push_back(lineInput);
        }

        return lines;
}

string trim (string input) {
        input.erase(remove(input.begin(), input.end(), ' '), input.end());
        input.erase(remove(input.begin(), input.end(), '\t'), input.end());
        return input;
}
```

B.2.4 Code Formatting Configuration File

The following is the configuration file used for Uncrustify, which is used by Light-PREM Source Analyzer. Due to the size of the configuration file, only parameters that were changed from their defaults are listed here.

```
# Uncrustify 0.59

# Add or remove newline between return type and function name in a function definition
nl_func_type_name                       = add      # ignore/add/remove/force

# Add or remove a newline between the return keyword and return expression.
nl_return_expr                          = remove   # ignore/add/remove/force

# Add or remove newline between 'if' and '{'
nl_if_brace                             = add      # ignore/add/remove/force

# Add or remove newline between '}' and 'else'
nl_brace_else                           = add      # ignore/add/remove/force

# Add or remove newline between 'else if' and '{'
# If set to ignore, nl_if_brace is used instead
nl_elseif_brace                         = add      # ignore/add/remove/force

# Add or remove newline between 'else' and '{'
nl_else_brace                           = add      # ignore/add/remove/force

# Add or remove newline between 'else' and 'if'
nl_else_if                              = remove   # ignore/add/remove/force

# Add or remove newline between 'for' and '{'
nl_for_brace                            = add      # ignore/add/remove/force

# Add or remove newline between 'while' and '{'
nl_while_brace                          = add      # ignore/add/remove/force
```

```
# Add or remove newline between 'do' and '{'
nl_do_brace                             = add      # ignore/add/remove/force

# Add or remove newline between '}' and 'while' of 'do' statement
nl_brace_while                          = add      # ignore/add/remove/force

# Add or remove newline between 'switch' and '{'
nl_switch_brace                         = add      # ignore/add/remove/force

# Newline between namespace and {
nl_namespace_brace                      = add      # ignore/add/remove/force

# Add or remove newline between return type and function name in a prototype
nl_func_proto_type_name                 = remove   # ignore/add/remove/force

# Add or remove newline between a function name and the opening '('
nl_func_paren                           = remove   # ignore/add/remove/force

# Add or remove newline between a function name and the opening '(' in the definition
nl_func_def_paren                       = remove   # ignore/add/remove/force

# Add or remove newline after '(' in a function declaration
nl_func_decl_start                      = remove   # ignore/add/remove/force

# Add or remove newline after '(' in a function definition
nl_func_def_start                       = remove   # ignore/add/remove/force

# Add or remove newline after each ',' in a function declaration
nl_func_decl_args                       = remove   # ignore/add/remove/force

# Add or remove newline after each ',' in a function definition
nl_func_def_args                        = remove   # ignore/add/remove/force

# Add or remove newline before the ')' in a function declaration
nl_func_decl_end                        = remove   # ignore/add/remove/force

# Add or remove newline before the ')' in a function definition
nl_func_def_end                         = remove   # ignore/add/remove/force

# Add or remove newline between '()' in a function declaration.
nl_func_decl_empty                      = remove   # ignore/add/remove/force

# Add or remove newline between '()' in a function definition.
nl_func_def_empty                       = remove   # ignore/add/remove/force

# Add or remove newline between function signature and '{'
nl_fdef_brace                           = add      # ignore/add/remove/force

# Whether to put a newline after 'return' statement
nl_after_return                         = true     # false/true

# Whether to put a newline after semicolons, except in 'for' statements
nl_after_semicolon                      = true     # false/true

# Whether to put a newline after brace open.
# This also adds a newline before the matching brace close.
nl_after_brace_open                     = true     # false/true

# Whether to put a newline after a virtual brace open with a non-empty body.
# These occur in un-braced if/while/do/for statement bodies.
nl_after_vbrace_open                    = true    # false/true

# Whether to put a newline after a virtual brace open with an empty body.
# These occur in un-braced if/while/do/for statement bodies.
nl_after_vbrace_open_empty              = true    # false/true

# Whether to put a newline after a brace close.
# Does not apply if followed by a necessary ';'.
nl_after_brace_close                    = true    # false/true

# Whether to put a newline after a virtual brace close.
# Would add a newline before return in: 'if (foo) a++; return;'
nl_after_vbrace_close                   = true    # false/true

# Other parameters
...
```

B.2.5 Cache Miss Analyzer

The following script is used to determine the number of cache misses executed by a predictable interval.

```
#!/bin/bash

if [ $# -lt 2 ]; then
        echo "Usage: prem_analyze <executable> <function name> [command line
args ...]"
        exit 1
fi

executable=$1
function=$2
shift 2
executable_options=$*

valgrind --tool=callgrind --cache-sim=yes --simulate-wb=yes --I1=32768,8,64 \
--D1=32768,8,64 --LL=131072,16,64 --toggle-collect=$function \
--callgrind-out-file=callgrind.out $executable $executable_options &> /dev/null

callgrind_annotate --auto=yes --show=DLmr,DLmw --threshold=100 callgrind.out > report

cg_format callgrind.out | grep "^[0-9][0-9]*,[0-9][0-9]*,[0-9][0-9]*," | grep \
-v PREM_ | sed 's/^\([0-9][0-9]*\),\([0-9][0-9]*\),\([0-9][0-9]*\),.*/\2/g' | \
awk '{s+=$1} END {print "read misses: " s}'

cg_format callgrind.out | grep "^[0-9][0-9]*,[0-9][0-9]*,[0-9][0-9]*," | grep \
-v PREM_ | sed 's/^\([0-9][0-9]*\),\([0-9][0-9]*\),\([0-9][0-9]*\),.*/\3/g' | \
awk '{s+=$1} END {print "write misses: " s}'

rm callgrind.out
```

B.3 Light-PREM Memory Analyzer Code

This section lists the source code of the Light-PREM Memory Analyzer.

B.3.1 Light-PREM Memory Analyzer Script

The following bash script runs Light-PREM Memory Analyzer. Note that the script calls other executables, some of which are included in this appendix as well, and some of which are not included.

```
#!/bin/bash

LIGHTPREM_DIR=/home/rtsl/svn/predictableCOTS/software/light-prem/tool2

if [ $# -lt 3 ]; then
        echo "Usage: lightprem <source file> <function name> <executable name> [executable arguments...]"
        exit 1
fi

SOURCE=$1
FUNCTION=$2
EXE_NAME=$3
shift 3

#check that we can compile with preprocessor
gcc $PREPROCESSOR_OPTS -E $SOURCE > /dev/null
RETVAL=$?
if [ $RETVAL -ne 0 ]; then
        echo "Error: Failed to run preprocessor. Please add preprocessor options"
        exit 1
fi

# generate formatted source file
gcc $PREPROCESSOR_OPTS -E $SOURCE | grep -v ^# | tr '\n' ' ' | sed 's/  */ /g' | \
uncrustify -c $LIGHTPREM_DIR/format.cfg | $LIGHTPREM_DIR/remove_func_bodies > formatted

# generate lightprem.h (profile version)
echo "/* automatically generated file */" > lightprem.h
echo >> lightprem.h

echo "#define PREM_BEGIN \\" >> lightprem.h
```

```
echo "  static LightPREM_count = 0; \\" >> lightprem.h
echo "  if (LightPREM_count == 0) { \\" >> lightprem.h
echo "          dump_mem_graph(\"lightprem.test.info\"); \\" >> lightprem.h
echo "  } \\" >> lightprem.h

echo "  fprintf(stderr, \"LightPREM start\\n\"); \\" >> lightprem.h
echo "  if (LightPREM_count == 0) { \\" >> lightprem.h
for name in `cat formatted | $LIGHTPREM_DIR/globals`; do
        echo "          fprintf(stderr, \"LightPREM $name %p\\n\", $name); \\" >> lightprem.h
done
for name in `cat formatted | $LIGHTPREM_DIR/locals $FUNCTION`; do
        echo "          fprintf(stderr, \"LightPREM $name %p\\n\", $name); \\" >> lightprem.h
done
echo "          LightPREM_count++; \\" >> lightprem.h
echo "  }" >> lightprem.h
echo >> lightprem.h

echo "#define PREM_END \\" >> lightprem.h
echo "  fprintf(stderr, \"LightPREM end\\n\");" >> lightprem.h
echo >> lightprem.h

rm formatted

# run executable, just with dump_mem_graph
make clean
make $EXE_NAME
RETVAL=$?
if [ $RETVAL -ne 0 ]; then
        echo "Error: Failed to compile profiling executable"
        exit 1
fi
echo "Running Executable with Memory Analysis..."
$EXE_NAME $@ &> /dev/null
echo "Run complete"

# modify where dump_mem_graph puts the file
sed -i 's/lightprem.test.info/lightprem.info/' lightprem.h

# run profiler
make clean
make $EXE_NAME
RETVAL=$?
if [ $RETVAL -ne 0 ]; then
        echo "Error: Failed to compile profiling executable"
        exit 1
fi

echo "Running Profiler: This may take a bit..."
valgrind --tool=lackey --trace-mem=yes $EXE_NAME $@ 2>&1 1>/dev/null | \
$LIGHTPREM_DIR/valgrind_filter 2> vars.info | sed 's/.*\([0-9a-f]\{8\}\).*/\1/' | sort -u >>lightprem.info
echo "Profiler run complete"

# generate lightprem.h (real version)
echo "/* automatically generated file */" > lightprem.h
echo >> lightprem.h

echo "#include \"prem_support.h\"" >> lightprem.h
echo >> lightprem.h

echo "#define PREM_END \\" >> lightprem.h
echo "PREM_END_PRED_INTERVAL(\"$FUNCTION\",0,0); " >> lightprem.h
echo >> lightprem.h

echo "#define PREM_BEGIN \\" >> lightprem.h
echo "PREM_BEGIN_PRED_INTERVAL(0); \\" >> lightprem.h
$LIGHTPREM_DIR/graph/gen_prefetch lightprem.info lightprem.test.info vars.info >> lightprem.h
echo "PREM_PREFETCH_STACK(sizeof(size_t), 0x0); \\" >> lightprem.h
echo "PREM_BEGIN_PRED_EXECUTION(\"$FUNCTION\",0); " >> lightprem.h
echo >> lightprem.h

rm lightprem.info
rm lightprem.test.info
rm vars.info

echo "Lightprem finished successfully"
```

B.3.2 Heap Analyzer

The following code is used by Light-PREM Memory Analyzer to intercept malloc calls, and is also used to print out connections between heap chunks to a file.

```
#define _GNU_SOURCE 1
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

void* (*get_malloc())(size_t) {
        static void *(*orig_malloc)(size_t) = NULL;
        if (orig_malloc == NULL) {
                orig_malloc = dlsym(RTLD_NEXT, "malloc");
        }

        return orig_malloc;
}

void * malloc(size_t size) {
        void *ptr = get_malloc()(size);

        add_ptr(ptr, size);

        return ptr;
}


void* (*get_free())(size_t) {
        static void *(*orig_free)(size_t) = NULL;
        if (orig_free == NULL) {
                orig_free = dlsym(RTLD_NEXT, "free");
        }

        return orig_free;
}

void free(void *ptr) {
        get_free()(ptr);
        remove_ptr(ptr);
}

#define assert(val) {                                                   \
        if (val == 0) {                                                 \
                fprintf(stderr, "Assertion failed on line %i\n", __LINE__); \
                exit(1);                                                \
        }                                                               \
}


/****** Book keeping of malloc chunk ******/

typedef struct malloc_chunk {
        void *ptr;
        size_t size;
        struct malloc_chunk *next;
} mchunk;

mchunk *mchead = NULL;
int nchunks = 0;

mchunk *new_node(void *ptr, size_t size) {
        mchunk *node = get_malloc()(sizeof(mchunk));
        node->ptr = ptr;
        node->size = size;
        node->next = NULL;
        return node;
}

inline int inside(void *ptr, size_t size, void *ptr2) {
        return ptr2 >= ptr && ptr2 < (ptr + size);
}

void add_ptr(void *ptr, size_t size) {
        if (size <= 0)
                return;

        if (mchead == NULL) {
                mchead = new_node(ptr, size);
                nchunks++;
                return;
        }

        mchunk *curr = mchead;
        while (curr->next != NULL) {
                // ensure no overlap
                assert(!inside(curr->ptr, curr->size, ptr) &&
                       !inside(ptr, size, curr->ptr));
```

```
                curr = curr->next;
        }

        curr->next = new_node(ptr, size);
        nchunks++;
}

void remove_ptr(void *ptr) {
        assert(mchead != NULL);
        if (mchead->ptr == ptr) {
                mchunk *n = mchead->next;
                get_free()(mchead);
                mchead = n;
                nchunks--;
                return;
        }

        mchunk *first = mchead;
        mchunk *second = mchead->next;
        assert(first && second);

        while (second->ptr != ptr) {
                first = first->next;
                second = second->next;
                assert(first->next == second);
                assert(second);
        }

        first->next = second->next;
        get_free()(second);
        nchunks--;
}

typedef struct mem_link {
        long add;
        long subtract;
        struct mem_link *next;
} mlink;

void dump_mem_graph(char *filename) {
        mlink *links[nchunks][nchunks];

        // initialize all links to empty
        int i, j;
        for (i = 0; i < nchunks; i++)
                for (j = 0; j < nchunks; j++)
                        links[i][j] = NULL;

        // create links (probing algorithm/"fishing for pointers")
        mchunk *curr;
        int index1;
        for (curr = mchead, index1 = 0; curr != NULL; curr=curr->next, index1++) {
                void *ptr;
                for (ptr = curr->ptr; ptr < curr->ptr + curr->size - sizeof(void*)+1; ptr+=sizeof(void*)){
                        void *value = *((void **) ptr);

                        mchunk *comp;
                        int index2;
                        for (comp = mchead, index2=0; comp != NULL; comp=comp->next, index2++) {
                                assert(index1 < nchunks && index2 < nchunks);

                                if (inside(comp->ptr, comp->size, value)) {
                                        mlink *new_mlink = get_malloc()(sizeof(mlink));
                                        new_mlink->add = ptr - curr->ptr;
                                        new_mlink->subtract = value - comp->ptr;
                                        new_mlink->next = NULL;

                                        if (links[index1][index2] == NULL)
                                                links[index1][index2] = new_mlink;
                                        else {
                                                mlink *mptr = links[index1][index2];
                                                while (mptr->next != NULL)
                                                        mptr = mptr->next;
                                                mptr->next = new_mlink;
                                        }
                                }
                        }
                }

        }

        /* opening a file will call malloc itself. Need to ignore those chunks
         * allocated by fopen's malloc, which we can do by keeping track of the
         * old value of nchunks */
        int old_nchunks = nchunks;
```

```
        // open file
        FILE *fp = fopen(filename, "w");
        if (fp == NULL) {
                fprintf(stderr, "Error: Could not open file \"%s\" for writing\n", filename);
                return;
        }

        fprintf(fp, "%i\n", old_nchunks);

        // write each malloc chunk
        for (curr = mchead, index1 = 0; curr != NULL && index1 < old_nchunks; curr = curr->next, index1++)
                fprintf(fp, "%p %p\n", curr->ptr, curr->ptr + curr->size);

        // write the links
        for (i = 0; i < old_nchunks; i++)
                for (j = 0; j < old_nchunks; j++) {
                        mlink *mptr = links[i][j];
                        mlink *tmp;
                        while (mptr != NULL) {
                                fprintf(fp, "%i %i %li %li\n", i,
                                                j, mptr->add, mptr->subtract);
                                tmp = mptr;
                                mptr = mptr->next;
                                get_free()(tmp);
                        }
                }

        fprintf(fp, "----\n");
        fclose(fp);
}
```

B.3.3 Prefetch Generator

The following code is used by Light-PREM Memory Analyzer to gather provided information from profiling and generate PREM prefetch code.

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <queue>
#include <vector>
#include <set>
#include <algorithm>
using namespace std;

#include "graph.h"

struct Node {
        void *start;
        void *end;
        bool isHandle;
        string name;

        Node(): start(NULL), end(NULL), isHandle(false), name("") {}
        Node(void* start, void* end): start(start), end(end), isHandle(false), name("") {}
        Node(string name, void* value): start(value), end(value), isHandle(true), name(name) {}
};

struct Edge {
        // edge from one Node node1 to another Node node2, such that:
        //      *((void**)(node1.start + add)) - subtract == node2.start
        long add, subtract;

        Edge(): add(-1), subtract(-1) {}
        Edge(long add, long subtract): add(add), subtract(subtract) {}

        bool operator==(const Edge &other) const {
                return other.add == add && other.subtract == subtract;
        }
};

struct Path {
        bool isValidPath;
        Node startNode;
        Node endNode;
        void *targetAddress;
        int size;
        vector<Edge> edges;
```

```
        Path(): isValidPath(false) { }
        Path(Node node, void *address):
                isValidPath(true), startNode(node), targetAddress(address), size(sizeof(void*)) { }
};

const int MAX_DEPTH = 10;


inline bool isBetween(void *ptr, void *start, void *end) {
        return ptr >= start && ptr < end;
}

inline bool isBetween(void *ptr, const Node & node) {
        if (node.isHandle)
                return false;
        return isBetween(ptr, node.start, node.end);
}

bool compFunc(pair<Edge, int> pair1, pair<Edge, int> pair2) {
        return pair1.first.subtract > pair2.first.subtract;
}

vector< pair<Edge, int> > filterAndSortEdges(void *address, const Node &node,
                                        const vector< pair<Edge, int> > & allEdges) {
        vector< pair<Edge, int> > result;
        vector< pair<Edge, int> >::const_iterator it;

        for (it = allEdges.begin(); it < allEdges.end(); it++) {
                long offset = it->first.subtract;
                if (((char*) address) >= ((char*) node.start) + offset)
                        result.push_back(*it);
        }

        sort(result.begin(), result.end(), compFunc);

        return result;
}

Path getBestPath(void *address, int curNodeIndex, const Graph<Node, Edge> & graph, int maxDepth) {
        Node curNode = graph.getNode(curNodeIndex);
        if (curNode.isHandle) {
                assert(address == curNode.start);
                return Path(curNode, address);
        }

        if (maxDepth <= 0)
                return Path();

        assert(isBetween(address, curNode));

        vector< pair<Edge, int> >  allEdges = graph.getOutgoingEdges(curNodeIndex);
        vector< pair<Edge, int> > possEdges = filterAndSortEdges(address, curNode, allEdges);
        vector< pair<Edge, int> >::iterator it;

        for (it = possEdges.begin(); it < possEdges.end(); it++) {
                Edge edge = it->first;
                int otherNodeIndex = it->second;
                Node otherNode = graph.getNode(otherNodeIndex);
                void *otherAddress = (void*) (((char*)otherNode.start) + edge.add);
                assert(((char*)curNode.start) + edge.subtract <= ((char*)address));

                Path path = getBestPath(otherAddress, it->second, graph, maxDepth - 1);

                // found a path!
                if (path.isValidPath) {
                        path.edges.push_back(it->first);
                        path.targetAddress = address;
                        path.endNode = curNode;
                        return path;
                }
        }

        return Path();
}

Path getBestPath(void *address, const Graph<Node, Edge> & graph) {
        const vector<Node> nodes = graph.getNodes();

        for (int i = 0; i < nodes.size(); i++) {
                if (isBetween(address, nodes[i]))
                        return getBestPath(address, i, graph, MAX_DEPTH);
        }

        return Path();
```

```
}

void printPrefetch(const Path &path) {
        Node startNode = path.startNode;
        Node endNode = path.endNode;
        vector<Edge> edges = path.edges;
        void *address = path.targetAddress;
        int size = path.size;
        long offset = 0;

        assert(path.isValidPath);
        assert(startNode.isHandle);
        assert(isBetween(address, endNode));

        string statement = string("(void*) ") + startNode.name;

        vector<Edge>::iterator it;
        bool first = true;
        for (it = edges.begin(); it < edges.end(); it++) {
                Edge edge = *it;
                long add = edge.add - offset;
                offset = edge.subtract;
                assert(add >= 0);

                ostringstream addString;
                addString << add;
                if (first) {
                        assert(add == 0);
                        first = false;
                } else {
                        statement = string("*(void**)(") + statement + " + " + addString.str() + ")";
                }
        }

        long finalDiff = ((char*)address) - (((char*)endNode.start) + offset);
        assert(finalDiff >= 0);
        ostringstream addString, sizeString;
        addString << finalDiff;
        sizeString << size;

        statement = "PREM_PREFETCH_GLOBAL( ((" + statement + ") + " + addString.str() + "), " +
                sizeString.str() + "); \\";

        cout << statement << endl;
}

vector<Path> combinePaths(vector<Path> &paths) {
        vector<Path> result;
        vector<Path>::iterator it;
        Path accPath;

        for (it = paths.begin(); it < paths.end(); it++) {
                Path path = *it;
                if (!accPath.isValidPath) {
                        accPath = path;
                        continue;
                }

                if (path.edges == accPath.edges &&
                   ((char*)accPath.targetAddress) + accPath.size  >= (char*) path.targetAddress) {
                        accPath.size = ((int) ((char*)path.targetAddress -
                                        (char*)accPath.targetAddress)) + sizeof(void*);
                } else {
                        result.push_back(accPath);
                        accPath = path;
                }
        }

        assert(accPath.isValidPath || paths.size() == 0);
        if (accPath.isValidPath)
                result.push_back(accPath);

        return result;
}

int main(int argc, char **argv) {
        if (argc != 4) {
                cerr << "Usage: " << argv[0] << " <info file> <test info file> <var file>" << endl;
                return 1;
        }

        ifstream file(argv[1]);
        if (!file) {
                cerr << "Error: Could not open file \"" << argv[1] << "\" for reading" << endl;
                return 1;
```

```
}

ifstream testfile(argv[2]);
if (!testfile) {
        cerr << "Error: Could not open file \"" << argv[2] << "\" for reading" << endl;
        return 1;
}

ifstream varfile(argv[3]);
if (!varfile) {
        cerr << "Error: Could not open file \"" << argv[3] << "\" for reading" << endl;
        return 1;
}

string input;

int numChunks;
file >> numChunks;
getline(file, input);

Graph<Node, Edge> graph;

// read in malloc chunks
for(int index = 0; index < numChunks; index++) {
        getline(file, input);
        istringstream iss(input);

        void *start, *end;
        iss >> hex >> start >> hex >> end;

        graph.addNode(Node(start, end));
}

int testNumChunks;
testfile >> testNumChunks;
getline(testfile, input);
assert(testNumChunks == numChunks);
set<string> connections;

// skip malloc chunks, should be same as info file
for (int i = 0; i < testNumChunks; i++) getline(testfile, input);

// record all connections from test file
while(getline(testfile, input)) {
        if (input.compare("----") == 0)
                break;
        connections.insert(input);
}

// read in connections between malloc chunks
while(getline(file, input)) {
        if (input.compare("----") == 0)
                break;

        // this was a fluke! (since its not also in the test file)
        if (connections.find(input) == connections.end())
                continue;

        istringstream iss(input);
        int index1, index2;
        long add, subtract;
        iss >> index1 >> index2 >> add >> subtract;

        graph.addEdge(index1, index2, Edge(add, subtract));
}

// read in variable addresses
while(getline(varfile, input)) {
        if (input.empty())
                break;

        istringstream iss(input);
        string name;
        void *address;
        iss >> name >> hex >> address;

        vector<Node> nodes = graph.getNodes();
        for (int i = 0; i < nodes.size(); i++) {
                Node node = nodes[i];
                if (isBetween(address, node)) {
                        int index = graph.addNode(Node(name, address));
                        long diff = ((char*)address) - ((char*)node.start);
                        graph.addEdge(index, i, Edge(0,diff));
                }
        }
```

```
        }

        graph.precomputeOutgoingEdges();
        vector<Path> paths;
        vector<Path>::iterator it;

        // read in raw memory accesses, get paths
        while(getline(file, input)) {
                istringstream iss(input);
                void *address;
                iss >> address;

                Path path = getBestPath(address, graph);
                if (path.isValidPath)
                        paths.push_back(path);
        }

        // combine paths
        paths = combinePaths(paths);

        // print paths
        for (it = paths.begin(); it < paths.end(); it++) {
                printPrefetch(*it);
        }

        return 0;
}
```

B.3.4 Utility Graph Class

The following C++ class is used to help represent connections between heap chunks in Light-PREM Memory Analyzer. It is used above in the Prefetch Generator code.

```
#include <vector>
using namespace std;

template <class N, class E>
class Graph {
public:
        int addNode(N node);
        void addEdge(int index1, int index2, E e);

        const vector<N> & getNodes() const;
        vector<E> getEdges(int index1, int index2) const;
        N getNode(int index) const;

        void precomputeOutgoingEdges();
        vector< pair<E, int> > getOutgoingEdges(int index) const;

        vector< pair<E, int> > getIncomingEdges(int index) const;

private:
        void assertIndex(int index) const;

        vector<N> nodes;
        vector< vector < pair<E, int> > > edges;

        bool precomputedOutgoing;
        vector< vector < pair<E, int> > > outgoingEdges;
};

// code inlined below

#include <assert.h>

template <class N, class E>
int Graph<N,E>::addNode(N node) {
        nodes.push_back(node);
        edges.push_back(vector< pair<E, int> >());
        assert(nodes.size() == edges.size());

        precomputedOutgoing = false;

        return nodes.size() - 1;
}

template <class N, class E>
void Graph<N,E>::addEdge(int index1, int index2, E e) {
        assertIndex(index1);
```

```
        assertIndex(index2);

        edges[index1].push_back(pair<E, int>(e, index2));

        precomputedOutgoing = false;
}

template <class N, class E>
const vector<N> & Graph<N,E>::getNodes() const {
        return nodes;
}

template <class N, class E>
N Graph<N,E>::getNode(int index) const {
        assertIndex(index);
        return nodes[index];
}

template <class N, class E>
vector<E> Graph<N,E>::getEdges(int index1, int index2) const {
        assertIndex(index1);
        assertIndex(index2);

        vector<E> result;
        typename vector< pair<E, int> >::const_iterator it;
        for (it = edges[index1].begin(); it < edges[index1].end(); it++) {
                if (it->second == index2)
                        result.push_back(it->first);
        }

        return result;
}

template <class N, class E>
vector< pair<E, int> > Graph<N,E>::getIncomingEdges(int index) const {
        assertIndex(index);

        return edges[index];
}



template <class N, class E>
void Graph<N,E>::precomputeOutgoingEdges() {
        if (precomputedOutgoing)
                return;

        outgoingEdges.clear();

        for (int i = 0; i < nodes.size(); i++)
                outgoingEdges.push_back(getOutgoingEdges(i));

        precomputedOutgoing = true;
}

template <class N, class E>
vector< pair<E, int> > Graph<N,E>::getOutgoingEdges(int index) const {
        assertIndex(index);

        if (precomputedOutgoing)
                return outgoingEdges[index];

        vector< pair<E, int> > result;
        typename vector< pair<E, int> >::const_iterator it;
        for (int i = 0; i < edges.size(); i++) {
                for (it = edges[i].begin(); it < edges[i].end(); it++)
                        if (it->second == index)
                                result.push_back(pair<E, int>(it->first, i));
        }

        return result;
}


template <class N, class E>
void Graph<N,E>::assertIndex(int index) const {
        assert(index < nodes.size());
        assert(index < edges.size());
        assert(nodes.size() == edges.size());
}
```