The University of New South Wales

School of Computer Science and Engineering

Profiling in Parallel to Reduce Overhead

Peter Allan Sankauskas (2250455)

Bachelor of Science (Computer Science)

June 2003

Supervisor: Jingling Xue
Assessor: Gernot Heiser

Abstract

Profiling a program carries with it an unavoidable overhead. Online optimisations that use this profile data produce increasingly better results the lower the level of overhead is. The less overhead that needs to be compensated for, the greater the speed up. Profiling in parallel is a new technique developed for the Intel Itanium family of processors that spans instrumentation over several basic blocks. By spanning the instrumentation code over more of the original instructions, the level of parallelism between the original code and the instrumentation is increased. Using a relatively conservative approach to profiling in parallel, it is possible to reduce the amount of overhead required to build a profile of a program by an average of 13.7%. Not only is this a method for reducing profiling overhead, it is also the groundwork for future research in the area of parallel execution on a single CPU.

Contents

Chapter 1

Introduction

Just as Moore's Law states, processing power has continued to double every 18 months. The speed of main memory on the other hand is increasing at the much slower rate of around 10% per year. The unfortunate outcome is an ever-increasing gap in performance between the CPU and main memory. This performance gap has inspired computer architects to insert smaller and faster cache memories between the CPU and main memory. One of the first processors to have a level 1 (L1) cache on die, at least from the Intel family, was the 80486DX released in April of 1989. It had a 8Kb 4-way set associative unified cache. Compare this to the more recent Intel Itanium 2 processor, which has three levels of cache – L1I: 16Kb, L1D: 16Kb, L2: 256Kb and L3: 1.5Mb or 3Mb. Cache memory only eliminates the processor-memory gap if the data being referenced is in the cache. With caches being much smaller than main memory, and the average size of a program increasing, caches are a precious resource.

To reduce the number of cache misses, two main approaches have been devised. The first approach is to focus on the cause of the problem: poor locality. When a program references data that is stored close together, there is a much greater chance that all of the data will be in the cache. Rearranging data layout and changing data access patterns can greatly improve the reference locality of a program. For regular code, such as array, this method is quite useful, as the length of an array is known at compile time. It is the compiler's task to layout the data in the best way to improve locality.

Many general-purpose programs use dynamic, pointer based data structures such as trees, graphs and linked-lists. These structures can grow and shrink in size as the program executes, and the data can exist anywhere on the heap. Since the size and formation of these data structures is not known at compile time, the above method cannot be applied. The consequence of this is these data structures usually suffer from poor locality. A cache-conscious data structure framework [Chi99] has been proposed

to improve the locality of these irregular data structures, though this does have an overhead each time a structure is altered.

The other approach is to focus on the problem: memory latency. Prefetching is a strategy that does not remove the number of cache misses but rather, causes a cache miss to occur before the data is required. There is no need to wait for the data to be copied from main memory to the cache due to the fact this operation was performed earlier, effectively removing memory latency. Therefore, when the data is referenced, the processor can function at full speed.

Profiling is a method where meta-data is collected while a program is executing. Meta-data is data about data, and in this case it describes when, where and how often specific data references occur. A profile is created by inserting code to collect meta-data into the program. The program is then executed with input that is representative of how the program will be used when complete. Once the program has finished executing, the profile can then be analysed.

There are many different ways to analyse profiling data. Possibilities include finding out how often a function is called; how much time is spent in a section of code; and finding data that frequently causes cache misses. Actions are then taken to optimise the problem code, improving its overall performance. Optimisations based on profiling data can be applied in two forms – static and dynamic. Static (or offline) optimisation is performed after the profile has been constructed, and the program has finished executing. Possible uses of this profile include altering data access patterns and data layout, and insert prefetching code for hot data streams. Since the program is no longer executing, several optimisation opportunities are lost. Information gathered on dynamic data structures is only valid while the program is executing. Once the program is terminated, these data structures cease to exist, and generally do not form the same structure in subsequent executions making them very difficult if not impossible to optimise statically. Static optimisations are only useful for operations that are constant each time a program is executed.

Dynamic (or online) optimisation is a process where a program optimises itself while it is executing. For example, a program could perform profiling of its data structures, analyse this profile, and insert prefetching code into itself to improve performance. Although this sounds like a great idea, it comes at the cost of overhead. Profiling and analysis will increase the execution time of a program. However, if the speed up in performance is greater than the increase from overhead, this is a worthwhile procedure. This process of profiling, analysing and optimising could be repeated many times during a program's lifetime, continuously improving its performance.

To reduce the amount of overhead consumed by profiling, a bursty tracing framework [CH02, HC01] has been designed. A bursty trace is a small but representative

subsequence of all events during execution. This is an extension of the Arnold-Ryder framework [AR01]. Bursty tracing uses edge profiling to profile only a representative sample of meta-data that is much easier to manage and analyse. By not recording meta-data for every access, overhead is reduced to 3-18%, which is acceptable for use in online optimising.

The decision of what to prefetch and when to prefetch is quite complex. Data must be prefetched early enough to not incur any memory latency, so depending on where the required data is located, the time for the L1 cache to be filled ranges from a few cycles (L2) to a few milliseconds (secondary storage). There is also a trade-off of what to prefetch. If too many references are prefetched, this could lead to cache contamination – a state in which the cache is polluted with data that is not required in the near future, and data that is required has been removed. To prevent cache contamination, only hot data streams should be prefetched [CH02].

To distinguish what is a cold data stream and what is a hot data stream, some analysis must be performed on the profiled data. The Sequitur algorithm [NMW97a] can be used to produce a context free grammar (CFG) of $\langle program\_counter,\ address \rangle$ pairs that represent the profile in a hierarchical format. The Sequitur algorithm was originally designed for compression, but its ability to detect repetitions within a sequence of symbols makes it a perfect algorithm for our purposes. These repetitions are important because they may represent data references that could be prefetched.

This new representation of the profile is then analysed by another algorithm [CH02] to detect hot data streams. It does this based on the number of uses and the length of a non-terminal. Code can then be constructed and added to detect the start of a hot data stream, and to prefetch the tail of the stream. Care must be taken when drawing the line between detection and prefetching of a stream. If too little of the stream is detected, the accuracy of the prefetching will decline. If too much of the stream is detected, prefetching opportunities are lost, and performance will not be optimal.

This technique of online profiling and prefetching still carries with it an overhead. This thesis proposes an extension to this technique in which as much of profiling as possible is performed in parallel with the original program. This will reduce the overhead of profiling since instructions are preformed in parallel rather than serial. The Itanium processor has what is known as *instruction groups*. These are groups of three instructions that can be performed simultaneously (as long as the processor has the resources to do so). Each cycle, the Itanium executes an entire instruction group. If the instructions cannot be found to execute in parallel, `nop` instructions are bundled to make up the three instructions. Since the average programmer does not write code to be executed in parallel, the majority of general-purpose applications will not utilise all three instructions. A compiler's instruction scheduler can find instructions that

can be parallelised, but these opportunities may not always be present. Performing online optimisations is a good way to utilise the full performance capabilities of the Itanium processor.

The technique proposed is not to modify the instruction scheduler, but rather to span instrumentation code over several basic blocks. Basic blocks usually act as barrier to code. Optimisations that move an instruction outside of a basic block require a lot of analysis and checking to ensure no dependencies are violated. Since the instrumentation instructions are already known, it is possible to place these instructions at places that provide better opportunities for the scheduler to parallelise the instrumentation with the original program. Executing profiling in parallel will reduce the overhead usually associated with profiling. Online optimisations that then use this profiling data will have to recover much less time than with traditional profiling techniques.

This thesis is organised in the following way. Chapter 2 provides the basis for this thesis by summarising background knowledge. Chapter 3 details the advantages of profiling in parallel and describes the instrumentation techniques used. Chapter 4 explains how these ideas are implemented, with the experimental results of the implementation presented in Chapter 5. These results are then discussed in Chapter 6. The ideas introduced in this thesis can be extended in many ways and have numerous applications which are illustrated in Chapter 7. An overall summary and conclusion are presented in Chapter 8.

Chapter 2

Background

This chapter provides the background knowledge needed to understand the motivation, ideas and techniques used throughout this thesis. Control Flow Graphs are referred to in almost every chapter, so this concept is introduced here with an example. Dynamic hot data stream prefetching (and the bursty tracing framework that it builds on) are also explained in detail. These two ideas build a profile of a program's accesses to memory. Since it is quite easy for a program to access memory frequently, these profiles can be huge. An algorithm named Sequitur can be used to compress this profile. The advantage of using Sequitur is that it also enables quick and easy detection of frequently occurring memory references.

Parts of these topics are used and/or re-implemented in this thesis so it is imperative to have at least some knowledge of how these ideas work. Details of which parts are used are noted in subsequent chapters.

2.1 Control Flow Graphs

A program can be split up into a number of units. At the top most level is the entire program. Each program is made up of one or more functions. Functions are made up of a number of basic blocks, and each basic block is made up of instructions. Whenever a branch is encountered, a new basic block is formed. Using this method of splitting up a program, it is possible to construct a control flow graph (CFG). An example of a CFG is shown in Figure 2.1. Each function in a program is a new region of code, with its own separate CFG. All functions and CFGs in this thesis are single entry, multiple exit, meaning that the path of execution must start at one specific basic block (the start block), but may end in several different basic blocks (end blocks). Most of the edges in a CFG are forward edges. Loops in the code form what is known as a back-edge, which is an edge to the start of a loop. The reason they are labelled back-edges

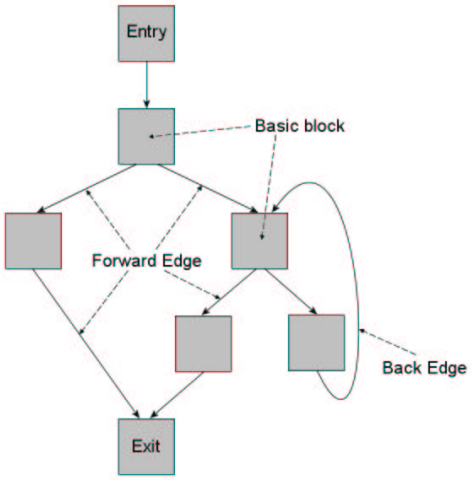


Figure 2.1: An example of a Control Flow Graph (CFG)

is because they are returning to a basic block that has already been executed. For a much greater explanation of CFGs, please refer to [Muc97].

2.2 Bursty Tracing Framework

The bursty tracing framework [HC01] was developed for low-overhead temporal profiling. It is an extension to the Arnold-Ryder framework [AR01] which constructs a temporal profile by sampling sub-sequences of the trace of all runtime events. The profiling information obtained from this framework is interprocedural, context-sensitive and flow-sensitive. It also does not require any operating system or hardware support.

The reason most profiling code has a high overhead is because all events during execution are recorded. This is known as a *trace*. To reduce overhead, it is possible to sample a small but representative fraction of the events, also known as a *burst*. In other words, bursty tracing is recording a subsequence of events captured by the trace of an executing program.

To do this, two copies of the program's code are made, both containing the original instructions. The first copy has a few additional instructions called *checks* located at loop-back edges and procedure entries. These instructions decrement a counter $nCheck$ which is initialised to $nCheck_0$. The second copy of the code is instrumented to collect profiling information. It also has similar checking code for $nInstr$. When the program is first started, it executes the checking code. Once $nCheck_0$ equals zero, $nInstr$ is initialised to $nInstr_0$ and the program executes the instrumented code. When $nInstr$ reaches zero, $nCheck$ is reset to $nCheck_0$ and the process continues.

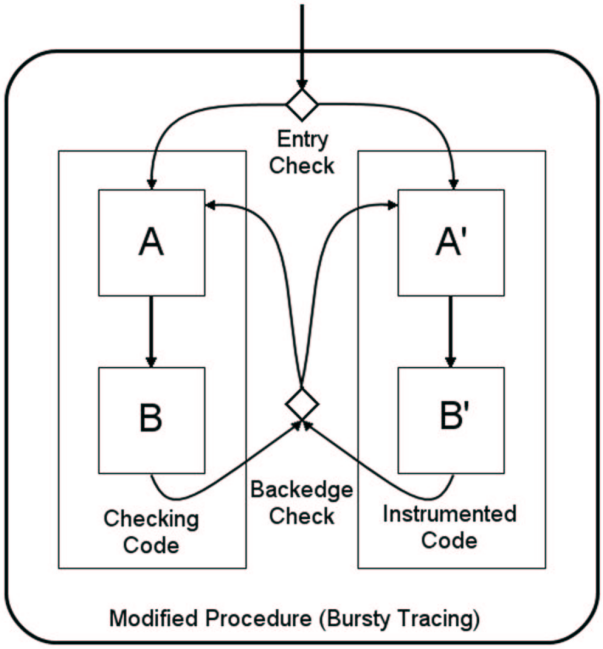


Figure 2.2: Bursty tracing framework.

Figure 2.2 displays this process in a graphical format.

The overhead for this framework is completely deterministic. Let $t_{checking}$ and $t_{instrumented}$ be the running time of the checking version of code and the instrumented version of code respectively. Also let $t_{original}$ be the running time of the original, unmodified code. The amount of overhead for a given configuration can be calculated by:

$$Overhead = \frac{nCheck_0 \cdot t_{checking} + nInstr_0 \cdot t_{instrumented}}{(nCheck_0 + nInstr_0) \cdot t_{original}} - 1$$

It is sometimes useful to refer to the sampling rate of this framework, as this determines the amount of overhead. This can be calculated using the following equation:

$$Sampling\ Rate = \frac{nInstr0}{nCheck_0 + nInstr_0}$$

To reduce overhead even further, not every procedure entry and loop-back edge has checking code. There only needs to be sufficient *checks* to ensure it is not possible for a program to recurse for an unbounded amount of time without executing a check. Based on a program's call graph, it is only necessary to insert checks at procedure entries when:

- the procedure is the root node in the call graph

- the procedure's address is taken as this may be part of recursion with indirect calls
- the procedure has recursion from below (ie, when it is called by a node further away from the root node than itself.).

It is also not necessary to insert checks inside tight inner loops. An example of a tight inner loop is a loop that compares two arrays, or copies arrays. These types of loops should be left to static optimisers. The threshold of when a loop should not have checks is defined by *k-boring loops* where there are no calls and less than or equal to k profiling events of interest. Experimental results in [HC01] state that eliminating *4-boring loops* improves the quality of the profile.

This framework is designed for use by online optimisations. As will be shown in Section 2.4, this can be extended to detect hot data streams, which can then be optimised.

2.3 Sequitur

Sequitur [NMW97a] is an algorithm developed by Craig Nevill-Manning and Ian Witten at the University of Waikato, New Zealand. Its purpose is to infer a hierarchical structure from a sequence of discrete symbols. It does this by forming a grammar based on repeated phases within a sequence. Each repeated phase forms a new rule in the grammar, and each use of that phase is replaced by the non-terminal symbol representing the phase. By removing repetitions, Sequitur produces a compressed representation of the original sequence.

Each grammar that Sequitur generates exhibits two properties:

digram uniqueness: no pair of adjacent symbols appear more than once in the grammar

rule utility: every rule is used more than once

These properties act as constraints for the Sequitur algorithm to enforce. Sequitur functions incrementally by taking a symbol and appending it to the starting rule. When another symbol is appended, these two symbols form a new **digram**. If this digram appears elsewhere in the sequence, the digram uniqueness constraint has been violated. At this point two actions can occur. If this is the first time the digram has been repeated in the sequence, a new rule must be created to replace it. If a rule already exists for this digram, then the digram is simply replaced by the rule's non-terminal symbol. Creating rules with more than two symbols is the side-effect of the rule utility constraint. Whenever a rule appears only once, it has violated the

rule utility constraint. To remove the offending rule, its digram is substituted where the single occurrence happens. This leaves the rule completely unused and is then removed from the grammar. Table 2.1 demonstrates how a grammar is created, and how the Sequitur algorithm enforced the two constraints at each step.

2.4 Dynamic Hot Data Stream Prefetching

Dynamic hot data stream prefetching [CH02] is a combination of the work presented in Section 2.2 and Section 2.3. It is a system that consists of three phases. The first phase is where a temporal data reference profile is constructed from an executing program. Second, the program is stopped while a fast analysis algorithm extracts hot data streams, and prefetching code is injected into the code. The final phase is termed hibernation, where the program executes with the added prefetching instructions, but without the profiling. Once the hibernation phase is complete, the inserted prefetching instructions are removed and the cycle starts again with the profiling phase.

A hot data stream is a sequence of data references that frequently repeat in the same order and are predictable. Programs usually contain a small number of hot data streams that account for approximately 90% of data references and more than 80% of cache misses [Chi01, SRC02]. These hot data streams consist of around 15-20 data references on average, which means they are long enough to be prefetched.

As mentioned earlier, this system has three phases – profiling; analysis and optimisation; and hibernation. The profiling phase makes use of the bursty tracing framework described in Section 2.2. This profiling phase is also termed the "awake" phase. Whenever $nCheck$ is reset to $nCheck_0$, the counter $nAwake$ is decremented. After $nAwake_0$ iterations, $nAwake$ will equal zero, signalling that the profiling phase is complete. This translates to having $nAwake_0 \cdot nInstr_0$ checks of traced data references in the profile.

The final phase is the hibernation phase, which lasts for $nHibernate_0$ iterations. At the beginning of each iteration, $nHibernate$ is decremented; $nCheck_0$ is set to $nCheck_0 + nInstr_0 - 1$; and $nInstr_0 = 1$. Setting $nCheck_0$ and $nInstr_0$ to these alternate values effectively stops the profiling code from being executed (except for one time each iteration). The reason for this is it ensures that one iteration, be it in the profiling phase or the hibernation phase, is the same length (measured in number of checks). Figure 2.3 displays the structure of the phases. This results in everything being deterministic and easy to control by manipulating the counters.

Dynamic Vulcan is used by [CH02] to make all of the modifications to the application. Vulcan [ASV01] is an x86 binary editor developed by Microsoft. When Vulcan modifies a binary, it makes a copy of the code and does the modifications to the copy.

Sequence	Grammar	Remarks
a	$S \rightarrow a$	
ab	$S \rightarrow ab$	
aba	$S \rightarrow aba$	
$abaa$	$S \rightarrow abaa$	
$abaab$	$S \rightarrow abaab$ ——— $S \rightarrow AaA$ $A \rightarrow ab$	ab appears twice enforce digram uniqueness
$abaabc$	$S \rightarrow AaAc$ $A \rightarrow ab$	
$abaabcd$	$S \rightarrow AaAcd$ $A \rightarrow ab$	
$abaabcda$	$S \rightarrow AaAcda$ $A \rightarrow ab$	
$abaabcdab$	$S \rightarrow AaAcdab$ $A \rightarrow ab$ ——— $S \rightarrow AaAcdA$ $A \rightarrow ab$	ab appears twice enforce digram uniqueness
$abaacdabc$	$S \rightarrow AaAcdAc$ $A \rightarrow ab$ ——— $S \rightarrow AaBdB$ $A \rightarrow ab$ $B \rightarrow Ac$	Ac appears twice enforce digram uniqueness
$abaacdabcd$	$S \rightarrow AaBdBd$ $A \rightarrow ab$ $B \rightarrow Ac$ ——— $S \rightarrow AaCC$ $A \rightarrow ab$ $B \rightarrow Ac$ $C \rightarrow Bd$ ——— $S \rightarrow AaCC$ $A \rightarrow ab$ $C \rightarrow Acd$	Bd appears twice enforce digram uniqueness B is only used once enforce rule utility

Table 2.1: Example of how Sequitur enforces constraints. Sourced from [NMW97a].

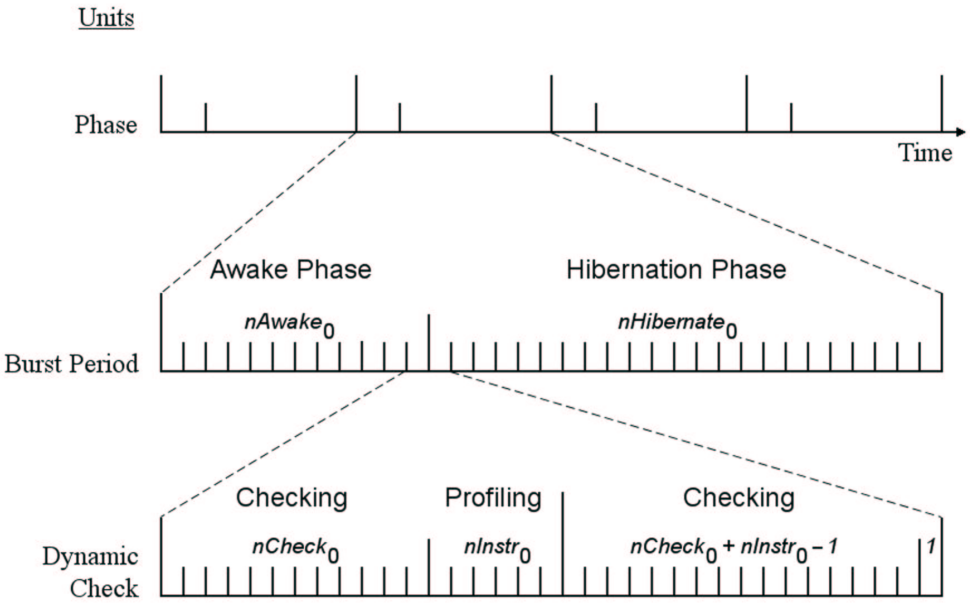


Figure 2.3: Structure of phases.

It then overwrites the first instruction of the original code with a `jump` to the modified code. When it comes time to de-optimise, all that needs to be done is replace the jump instruction with the original instruction. This modified binary file is then linked with the dynamic optimiser and analysis library.

2.4.1 Analysis

In order to analyse the profile data and detect hot data streams, the Sequitur algorithm detailed in Section 2.3 is used. Each profile event is a $\langle pc,\ addr \rangle$ pair and is treated as a single terminal symbol. This allows Sequitur to construct a context-free grammar from the profile. Since Sequitur is incremental, the profiling data collected is batched and sent to Sequitur as soon as it is collected rather than at the end of the profiling phase.

At the beginning of the analysis phase, the grammar is already present, and it is simply a matter of extracting the hot data streams. Each non-terminal symbol, A, in the grammar produces a sequence of data references, and as such has a heat value, $A.heat$, associated with it. The sequence that any non-terminal produces is labelled A_s, and its heat value is determined by:

$$A.heat = A_s.length \cdot A.coldUses$$

where $A.coldUses$ is "the number of times A occurs in the (unique) parse tree of the complete grammar, not counting occurrences in sub-trees belonging to hot non-terminals other than A. If the length of a hot non-terminal is to short, there isn't enough time for the prefetching to make a difference. On the other hand, if the length is too long, accuracy of the prefetching is decreased. Based on this, a non-terminal A is hot iff $minLen <= A.length <= maxLen$ and $A.heat >= H$, where H is a predetermined heat threshold".

The pseudo-code for the analysis framework is displayed in Figure 2.5. Lines 1-14 determine the order in which the non-terminals will be processed. It is simply the reverse post-ordering of the non-terminals based on the grammar's parse tree. Lines 15-22 set the number of times each non-terminal is used (ie. Calculates $coldUses$). Finally, lines 23-32 detect which non-terminals are hot, and lists them to be prefetched. This analysis framework is linear based on the number of non-terminals in the grammar.

For an example, let us extend the grammar presented in Table 2.1 to the grammar in Figure 2.4. Lets also make the following assumptions:

- heat threshold: $H = 11$
- minimum length restriction: $minLen = 3$
- maximum length restriction: $maxLen = 10$

When the analysis of this grammar is complete, the result is a single hot data stream constructed from the non-terminal B. The sequence B_s accounts for $16/19 = 84\%$ of all data references. The resulting values from the analysis are displayed in Table 2.2. Note that as B has two children, namely C and C, $B.uses$ is subtracted twice from $C.coldUses$ resulting in it being zero.

	Child	length	index	uses	coldUses	heat	Report?
S	A, B, B	19	0	1	1	19	no, start
A	-	2	3	5	1	2	no, cold
B	C, C	8	1	2	2	16	yes
C	A	4	2	4	0	0	no, cold

Table 2.2: Computed values from the hot data stream analysis.

2.4.2 Optimisation

Now that the analysis framework has detected all the hot data streams, code must be inserted into the program to optimise these accesses. Each hot data stream is partitioned into a head and a tail, with the idea being to detect the head of the sequence and prefetch the tail. The length of the head is determined by the fixed constant $headLen$. For example, we have a sequence $s = abacdef$ and let $headLen = 3$. The head of the sequence $s.head = aba$, and the tail is $s.tail = cdef$. Recall that each symbol is made up of a $\langle pc,\ addr \rangle$ pair. To keep track of how much of s has been detected so far, the variable $s.seen$ is associated with it. Once $s.seen == headLen$, it is time to prefetch $s.tail$. Code is inserted at $a.pc$ and $b.pc$ to detect $s.head$ and to prefetch $s.tail$. Pseudo-code for this process is detailed in Figure 2.6.

This technique works well for one sequence, but suppose we have multiple hot data streams with terminals that overlap. Having a separate $s.seen$ variable and independent checks for each stream introduces a lot of redundant checking code. This detection technique can be transformed into a deterministic finite state machine (DFSM) in which the states are the variable $s.seen$ and the transitions between states are the data references. Here is where a decision must be made: to have one DFSM for each sequence, incurring a high overhead in checking code, or have a combined DFSM for all hot data streams which reduces checking overhead, but comes at the cost of building the DFSM. Since the checking code is executed quite frequently, building one DFSM is the better option. Details of how to build this DFSM and the pseudo-code for it are depicted in [CH02]. They note that the number of states in the DFSM for the worst-case is $2^{headLen \cdot n}$ where n is the number of hot data streams. This might make it seem like constructing a single DFSM is not the best solution. However, in practice

$S \rightarrow AaBB$
$A \rightarrow ac$
$B \rightarrow CC$
$C \rightarrow Acd$

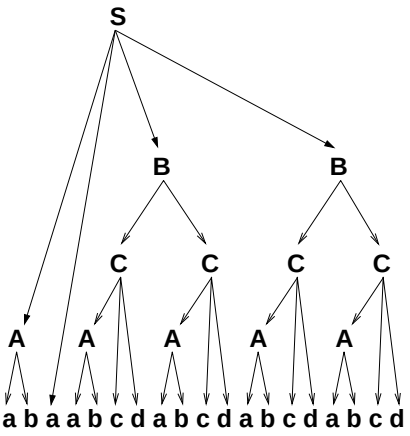


Figure 2.4: Extended grammar from Table 2.1 and its parse tree.

```
1   // Find reverse post-order numbering for non-terminals
2   // (also known as depth-first traveral I think)
3   int next = number of rules in grammar
4   doNumbering(NonTerminal A) {
5       if (have not yet visited A) {
6           foreach (child B of A)
7               doNumbering(B);
8           next--;
9           A.index = next;
10      }
11  }
12
13  doNumbering(S);
14
15  // Find uses for non-terminals, initialise to coldUses to uses
16  foreach (NonTerminal A)
17      A.uses = A.coldUses = 0;
18
19  S.uses = S.coldUses = 1;
20  foreach (NonTerminal A, ascending order of A.index)
21      foreach (child B of A)
22          B.uses = B.coldUses = (B.uses + A.uses);
23  // Find hot non-terminals
24  foreach (NonTerminal A, ascending order of A.index) {
25      A.heat = A_S.length * A.coldUses;
26      fHot = (minLen <= A.length <= maxLen) && (A.heat >= H);
27      if (fHot)
28          reportHotDataStream(A);
29      subtract = fHot ? A.uses : (A.uses - A.coldUses)
30      foreach (child B of A)
31          B.coldUses = B.coldUses - subtract;
32  }
```

Figure 2.5: Pseudo-code for analysis framework. Sourced from [CH02].

```
a.pc: if (accessing a.addr) {
          if (s.seen == 2) {
              s.seen = 3;
              prefetch c.addr, d.addr, e.addr, f.addr;
          } else
              s.seen = 1;
      } else
          s.seen = 0;

b.pc: if (accessing b.addr) {
          if (s.seen == 1)
              s.seen = 2;
          else
              s.seen = 0;
      } else
          s.seen = 0;
```

Figure 2.6: Detection and prefetching code for the sequence $abacdef$.

the number of states is usually close to $headLen \cdot n + 1$ which is acceptable.

Chapter 3

Profiling in Parallel

With the introduction of the Itanium processor, and Instruction Level Parallelism (ILP), it is becoming easier and easier to execute multiple instructions simultaneously. As things stand today, the average programmer does not write a program with parallelism in mind. In fact, the most popular programming languages such as C, C++ and Java do not even offer support for parallelism without additional packages. As a result, it is up to the compiler to parallelise as much of the code as possible. Some code is easier to parallelise than others, so for the code that cannot be parallelised, the full power of the CPU is not utilised. To remedy this problem, this thesis proposes these unutilised instructions be used to perform profiling, which can then be used to optimise the program.

Profiling usually carries with it an overhead – larger code size means greater loading time, and potentially more instruction cache misses; longer execution time as there are more instructions to be executed; larger memory usage; and combat between original program and profiling code for CPU resources. If the profiling code can be executed in parallel, then at least one of the contributors to the overhead – longer execution time – will be reduced if not removed completely.

3.1 Hardware

This thesis is implemented specifically for the Intel Itanium (IA-64) family of processors. The Itanium processor executes instructions a bundle at a time. An instruction bundle consists of three instructions for the processor to execute simultaneously. Of course, not any three instructions can be bundled together, there are a few restrictions. The three instructions must have no read after write (RAW) or write after write (WAW) dependencies. The other restriction is that the processor must have the resources to perform all three instructions simultaneously. For example, the processor

Cache	L1 Instr.	L1 Data	Level 2	Level 3
Size	16kb	16kb	256kb	3Mb or 1.5Mb
Associativity	4-way	4-way	8-way	12-way
Line size	64 bytes	64 bytes	128 bytes	128 bytes
Latency	1 cycle	1 cycle	Minimum 5 cycles	Minimum 12 cycles

Table 3.1: Cache specifications for an Intel Itanium 2 processor. Sourced from [Int02].

must have enough memory bandwidth to perform more than one load instruction at a time, and must have enough execution units to perform more than one integer or floating point instruction at a time.

The Itanium and Itanium 2 processors come in a variety of different models. The Itanium 2 processor for example has three levels of cache with the first level being split into an instruction cache and a data cache. Table 3.1 is a summary of the on-chip caches of the Itanium 2 processor.

The Level 1 data cache (L1D) has two dedicated load ports and two dedicated store ports. The L1D cache has a write through policy meaning that all stores are passed to the L2 cache. This is true of both the Itanium and Itanium 2 processor. If a store misses in L1D, no penalty is incurred because the data is passed to the L2 cache without the L1D cache being filled. Floating-point loads and stores are also passed to the L2 cache. No floating-point data is held in the L1D cache.

Given these circumstances, the profiling techniques used in this thesis are solely directed at the L1D cache. As such, only integer load instructions are profiled. For more information about the Itanium processor, please refer to [Int99, Int02, int].

3.2 Overhead

When profiling for offline optimisation, overhead is not an issue as this is not how the final program will execute. The profile data should be as accurate as possible to improve the offline optimisations. However, when profiling for use in online optimisation, overhead is of major concern. The extra time taken to do the profiling, analysis and optimisation must be recovered by the speed up in execution time. This point is made quite clear in [CH02]. They have developed an online optimisation technique with low overhead (typically 3-18%) that is a perfect candidate for use in a parallel setting. They used the bursty tracing framework [HC01] described in Section 2.2, which is in part re-implemented in this thesis.

The reason for using the bursty tracing framework is simple. It already has a low overhead, and is deterministic. It is also very easy to control by manipulating the

counters, giving a large amount of flexibility. However, this technique also brings with it an increase in code size by more than 200% since there are two copies of the original code, plus the profiling code. The only time this should impact on performance is loading time, and on transitions between checking code and instrumented code where a few instruction cache misses could occur.

3.3 Technique

The technique used to gather profiling data in not the standard method of profiling. Most profiling techniques count the number of times a function or basic block is executed. The instrumentation used in this thesis does not actually count anything, instead it records the address used by each load instruction and the value of the instruction pointer at that point. This is the method used in [CH02] and described in Section 2.4.

For each integer load instruction (`ld8` in assembly), the address that is loaded is stored as well as the current instruction pointer (IP). Note that in [CH02], the IP is referred to as the program counter (PC). This information will eventually be passed to the Sequitur algorithm (detailed in Section 2.3). To prevent the invocation of the Sequitur algorithm on each load, both the IP and the loaded address are stored in an array. Once that array if filled, the information is then passed to Sequitur, which in turn compresses the profiled data. Pseudo-code for this instrumentation technique is depicted in Figure 3.1. Altering the size of the array (`profile_array_size`) controls the rate at which the Sequitur algorithm is called. The larger the size, the less frequent Sequitur is performed.

```
     // Original program (load)
0    x = *(addr);

     // Instrumentation
1    tmp_ip = ip_register;
2    profile_array[current_elem].ip = tmp_ip;
3    profile_array[current_elem].addr = addr;
4    count++;

5    if (current_elem >= profile_array_size) {
6        Sequitur();
7        current_elem = 0;
8    }
```

Figure 3.1: High level pseudo-code for the instrumentation technique

3.4 Spanning Instrumentation Over Basic Blocks

Executing two almost unrelated code streams – original code and profiling code – in parallel seems simple enough. However, these two code streams will be executed on the same CPU. This is quite different than if the two streams were being executed on different processors. A CPU usually steps through code sequentially, executing whatever the instruction pointer (IP) is pointing to. The only time the IP is not incremented sequentially are branch instructions. This means that the original code and the profiling code must be bundled together so that neither stream causes part of the other stream to be missed.

The Itanium Instruction Set Architecture (ISA) has many different types of branch instructions including conditional branches (`br.cond`), function calls (`br.call`) and function returns (`br.ret`), as well as counted loop and modulo-scheduled loop branches. At these locations, the point of execution can move anywhere. The Itanium processor reads an instruction bundle containing three instructions from the address pointed to by the IP. Since the original code and the profiling code are being executed together, these points can cause problems.

There are three cases which need to be considered: no branch instructions, branches in the original code, and branches in the instrumentation code. The simplest case is no branch instructions which does not cause any problems. This does not mean the entire program has no branches. Such a program would be fairly pointless. It means that the number of cycles between a load and a branch instruction is large enough to insert the instrumentation code. Hence, the instrumentation is entirely contained within a basic block. Every branch instruction is the final instruction in a basic block.

3.4.1 Branches in the Original Code

Spanning instrumentation code over one or more basic blocks is not a trivial task. If the instrumentation code was added to the original code over a branch instruction, there is a possibility that part of the instrumentation code could be skipped. This situation occurs when there is a load instruction (`ld`) closely followed by a branch instruction (`br`). In other words, if instrumentation code spans more than one basic block, the code in the basic block that is not executed is skipped. Different types of branch instructions form different CFGs, and therefore must be treated differently.

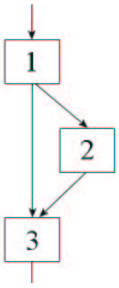
Conditional Branches

For conditional branches, there are a number of distinct cases which are shown in Figure 3.2. In every case, basic block 1 (BB:1) contains one load instruction, and

Case A:

An `if` with no `else`
Eg.

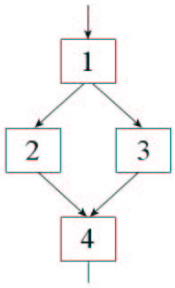
```
if (condition) {
    something
}
```



Case B:

Traditional `if-else` statement
Eg.

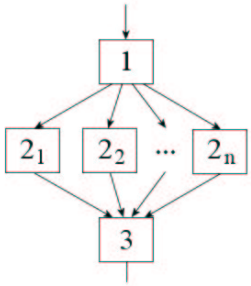
```
if (contition) {
    something
} else {
    something else
}
```



Case C:

A `switch` or `if-elseif-else` block
Eg.

```
switch (variable) {
    case 1: something_1; break;
    case 2: something_2; break;
     ...
    case n: something_n; break;
}
```



Case D:

A `do-while` loop
Eg.

```
do {
    something
} while (condition);
```

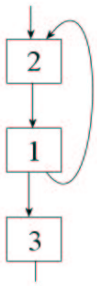


Figure 3.2: Control flow graphs cause by different conditional branches.

the final instruction in BB:1 is a conditional branch (`br.cond`). Let each basic block contain a total of seven instructions. Also, let the size of the instrumentation code be nine instructions, of which none are branch instructions. Using these two numbers, it is possible to create a ratio of instrumentation-to-original instructions, which is termed the *Instruction Ratio*. These conditions result in an Instruction Ratio of 9 : 7 which is too high for a likely chance of parallel execution between the two streams of code. Of course this depends exactly on the instruction in both streams, but for now this assumption will suffice. Dividing this ratio over two basic blocks gives 4 : 7 and 5 : 7 (named Stage 1 and Stage 2 respectively), which have a much better chance of parallelisation. To achieve these lower Instruction Ratios, the instrumentation code will have to span over one or more basic blocks.

Case A: Stage 1 of the instrumentation can be inserted into BB:1 without a problem. However, it is impossible to add Stage 2 into BB:2 because if BB:2 is not executed, part of the instrumentation code is skipped. Therefore, the remaining instrumentation instructions should be inserted into BB:3.

Case B: Stage 1 of the instrumentation code can be inserted into BB:1. Stage 2 can be duplicated in BB:2 and BB:3. This keeps both conditions of the branch consistent while maintaining a good chance of parallelisation.

Case C: Multi-way branches pose a problem. As usual, Stage 1 is inserted into BB:1. It is possible to duplicate Stage 2 along each path between BB:1 to BB:3, but this may result in a lot of duplicate code. Another solution is to add Stage 2 to BB:3. This solution is only possible if for all paths from BB:1 to BB:3, there are no load subsequent load instruction or call instructions (discussed below). If there are subsequent load instructions, the sequence of execution would be – Stage 1, Stage 1, Stage 2, Stage 2 – which will produce undefined results. The instrumentation technique used in this thesis cannot be nested within itself. It may be possible to design instrumentation code that can be nested within itself, but this is left for future work.

Case D: When this case occurs, there is no choice but to insert both Stage 1 and Stage 2 into BB:1. If Stage 2 were placed in BB:3, it would only be executed when the condition was false (end of loop). On the other hand, if Stage 2 were place in BB:2, then upon entrance to the loop, Stage 2 would be executed before Stage 1, which is incorrect. Stage 2 depends on Stage 1, they are not independent.

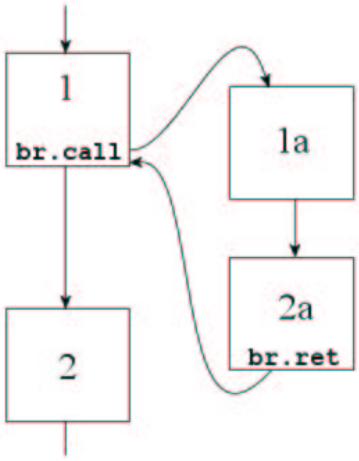


Figure 3.3: Control flow graph from call and return branch instructions.

Call and Return Branches

Call and return branches (`br.call` and `br.ret` respectively) act as barriers to instrumentation code. For return branches this is obvious since they mark the end of a region of code. The reason call branches act as barriers is simple. Suppose Stage 1 of the instrumentation has been executed and then a call branch is encountered. That call will invoke a new region of code, which has its own instrumentation code associated with it. After returning to the source of the call, Stage 2 would then be executed. The values attained from Stage 1 would have been invalidated by the instrumentation in the called region of code causing Stage 2 to either used invalid data or worse still, cause the program to halt. The result is that all instrumentation code must be completed before any call branch. Figure 3.3 is a pictorial view of the CFG created by call and return branches.

Loops

Any instrumentation inserted inside a loop structure must be completely contained within the loop. As stated in Case D of Conditional Branches, spanning instrumentation over loop boundaries produces incorrect results. Loops can be expressed in terms of **nesting**. The entry to a procedure is always the base level nesting (zero level of nesting). A loop within the base is the first level of nesting. A loop within the first level of nesting becomes the second level of nesting, and so on. Loops can be made up of several basic blocks, so each basic block has a nesting level associated with it. Therefore, the nest level of a basic block that contains a load instruction determines

the nest level for all basic blocks that may contain its instrumentation code.

3.4.2 Branches in the Instrumentation Code

Branches in the instrumentation code pose problems in a similar way as branches in the original code do – they may cause part of the original program to be skipped. With these branches, new basic blocks are created, basic blocks that did not exist in the original program. If parallelism between the original program and the instrumentation is to be maintained, part of the original program must be duplicated over these new basic blocks so that no matter what path is taken, no instructions are skipped.

Duplicating instruction and added basic blocks may sound simple enough, but think about when this will occur. If there are branches in the profiling code, these branches will be added for every load instruction in the original program. This may result in more than double the number of original basic blocks.

A much better solution is to design the instrumentation so that it does not produce any new basic blocks. This means there cannot be any branch instructions of any kind in the instrumentation. This is very limiting, so the resolution is to only permit call branches. Call branches only alter the region of code being instrumented by causing the basic block to be split in two. The instrumentation code presented in Figure 3.1 contains one conditional branch instruction (line 5). If this instruction is changed to a single call to the Sequitur algorithm, all instrumentation code will be straight-line code. For purpose of this thesis, a call is considered straight-line code. Even though it branches off, it still returns to the point of call, so there is no chance of instructions being skipped.

This does not change the fact that the Sequitur function will be called for every load instruction. Originally, the Sequitur algorithm was only executed when the array used to temporarily store the profile (`profile_array`) was full, and after executing the Sequitur algorithm, `current_elem` was reset to zero. The solution is to make the Sequitur function do a little more work than it used to. Resetting `current_elem` to zero can be moved to the final instruction in the Sequitur function. At the beginning of the Sequitur function, there can be a condition to immediately return if the array is not full. If it is full, it processes the array as per usual. In other words, the conditional branch of code is removed from the profiling code and placed in the Sequitur function. This translates to the instrumentation code being free of conditional branches, allowing it to be spanned over several basic blocks just as Subsection 3.4.1 describes. The low level assembly code for this instrumentation is shown in Figure 3.4

This only solves one problem, and creates another. It solves the complexity of trying to parallelise branches in the instrumentation by moving them to a function

```
 0    ld8       load_val   = [load_addr]      ; Original program
 1    mov       ip_reg     = ip               ; From here down is
                                               ; instrumentation

 2    addl      pa_lkup    = @ltoff(profile_array#), gp
 3    ld8       pa_addr    = [pa_lkup]

 4    addl      ce_lkup    = @ltoff(current_elem#), gp
 5    ld8       ce_addr    = [ce_lkup]
 6    ld4       ce_val     = [ce_addr]

 7    adds      r1         = 1, ce_val
 8    st4       [ce_addr]  = r1

 9    sxt4      r2         = ce_val
10    shladd    r3         = r2, 4, pa_addr
11    st8       [r3]       = ip_reg
12    adds      r4         = 8, r3
13    st8       [r4]       = load_addr

14    br.call.dpnt.few Sequitur
```

Figure 3.4: Assembly instruction for the instrumentation technique

that is completely separate from the original program. It creates the problem of a function call for each load. Traditionally function calls require spilling and filling of registers. The Itanium has a register stack engine (RSE) that removes this problem assuming that no more than 96 registers are in use at any one time.

A function call for each load, even if it does only do a compare and then exit the majority of the time, is still an overhead. It may be possible to reduce the number of times the function is called by analysing the original program. For example, only call Sequitur at the start and end of each region, and have an overflow array so that if Sequitur is not called for a substantial amount of time, the array is not filled beyond capacity. This reduction in calls is left for future work.

3.4.3 Splitting Instrumentation

Splitting the instrumentation into stages that can be spanned over basic blocks has been frequently mentioned in this chapter, but details of how to split instrumentation have been avoided. The reason for introducing this latter rather than earlier, is that to be able to completely explain the technique, a solid example is needed. Figure 3.4 provides the code for this example.

A quick analysis of this code shows that there are several dependencies between the instructions. For example, line 3 depends on the result of line 2; line 6 depends on line 5, which depends on line 4. These two sets of dependencies (lines 2-3 and 4-6) have no common dependencies. This means it is possible to execute lines 2 and 4 simultaneously, just as it is possible to do the same with lines 3 and 5. This is the type of parallel execution we are looking for between the original code and the instrumentation code.

The first step in splitting the instrumentation into stages is to remove the instructions that always produce the same result. These instructions should be executed once at the beginning of a region of code and their results stored in a register for the duration of that region. Since the Itanium processor has an abundance of registers (128 64-bit general purpose registers), keeping one or two registers aside should not impact on the register allocation phase too much. After examining the instrumentation, it is clear that lines 2-5 always produce the same results. Lines 2 and 4 lookup the address of a symbol, and lines 3 and 5 load the address of that symbol. The results of lines 2 and 4 are not used anywhere else in the instrumentation, so they do not need to be stored throughout the entire region, only the results of the two loads must be stored. This requires two dedicated registers for each region of code. These four instructions can now be removed from the main instrumentation block and be performed once upon entrance to the region.

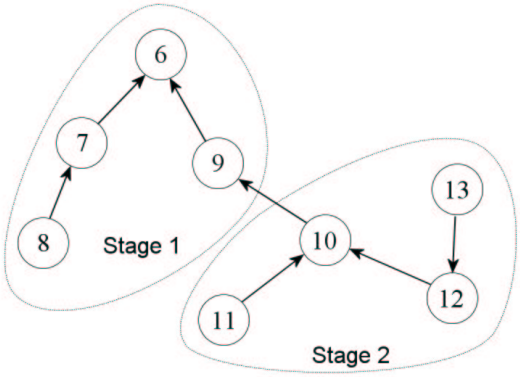


Figure 3.5: Dependence graph for main instrumentation block. Nodes are line numbers from Figure 3.4.

We now have line 1 and lines 6-14 to split into stages. Line 14 is a call branch to Sequitur, and as such must be the final instruction in a basic block. This line can also be removed from the main instrumentation block. For a similar reason, line 1 which must be bundled with the load being instrumented can be removed from the main instrumentation block. This leaves lines 6-13 that need to be split into stages.

To have a high chance of parallelisation between the original code and the instrumentation code, the instrumentation code should not be able to satisfy its dependencies too quickly. Figure 3.5 shows a dependence graph for the remaining eight instructions in the main instrumentation block. The instrumentation techniques described above require at most two stages. Since there are eight instructions, two stages of four instructions each can be created. From the dependence graph, Stage 1 should consist of lines 6-9 while Stage 2 consists of line 10-13. This results in only 2 of the 4 instructions in each stage being able to be executed in parallel.

It is possible of create a stage where no instructions can be executed in parallel. Such a stage would contain lines 6, 9, 10 and 11. All of these instructions depend on each other. This would mean that the other stage would consist of 7, 8, 12 and 13. This stage is not as favourable because both 7 and 12 can be parallelised, as can 8 and 13. The original segmentation of Stage 1 and Stage 2 from the previous paragraph is the most fair and is the split used in the implementation.

3.5 Summary

This chapter has introduced many new ideas, so the main points are summarised here. This instrumentation technique, profiling in parallel, has been created specifically for

the Intel Itanium family of processors, and is directed solely at the L1 data cache. Profiling carries with it an unavoidable overhead. Online optimisations that use this profile produce increasingly better results the lower the amount of overhead is. The less overhead that needs to be compensated for, the greater the speed up.

The traditional way of instrumenting code is to place it entirely in one basic block. Profiling in parallel introduces a new technique of spanning the instrumentation code over several basic blocks. The idea behind this is that spanning the instrumentation out over more of the original instructions will increase the level of parallelism between the original code and the instrumentation.

Special care must be taken when spanning instrumentation. Certain situations caused by branches and loops can force instrumentation to be restricted to a single basic block. In this case, there is no choice but to instrument that section of code using the traditional approach. For the sections that can be parallelised, the instrumentation code must be split into stages. Splitting instrumentation can be done by analysing its dependencies and selecting instructions that have the best chance of being spread over a basic block. These steps are the basis of profiling in parallel.

Chapter 4

Implementation

The ideas presented in Chapter 3 have little meaning if they are not evaluated and tested. To be able to do this, these ideas need to be implemented. There are two main concepts than required implementation, the Sequitur algorithm and the profiling methodology. The implementation of Sequitur must be tailored to process a rather unorthodox grammar constructed from symbols made up of a pair of addresses, namely, the address of the load instruction and the instruction pointer where the load occurs. After this is implemented, the instrumentation techniques discussed in Chapter 3 need to be implemented. Once both sections are complete, they can be used together to construct a compressed profile of a program's memory accesses, which in turn, can be analysed to find hot data streams.

4.1 Sequitur

The Sequitur algorithm has been implemented using the same method as described in [NMW97a, NMW97b]. To keep the code well structured, C++ was chosen because of the need for classes and inheritance. The grammar is constructed from a series of interconnected nodes. At the lowest level, there are three node classes, Terminal, Non-Terminal and Guard, which all extend from a generic node class. An instance of the Terminal class holds the IP and load address for a specific load instruction. Non-Terminal nodes are simply a pointer to a rule. A Rule contains one Guard node and a number of Terminal and/or Non-Terminal nodes. All nodes are connected by a doubly linked list, with the start and end of each list being a single Guard node. Figure 4.1 shows the structure created by this implementation.

To speed up insertion and deletion of nodes, a hash table of digrams is used. This table has one entry in it for each combination of Terminals and Non-Terminals. This way, there is no need to parse the entire grammar to see if a digram is already used.

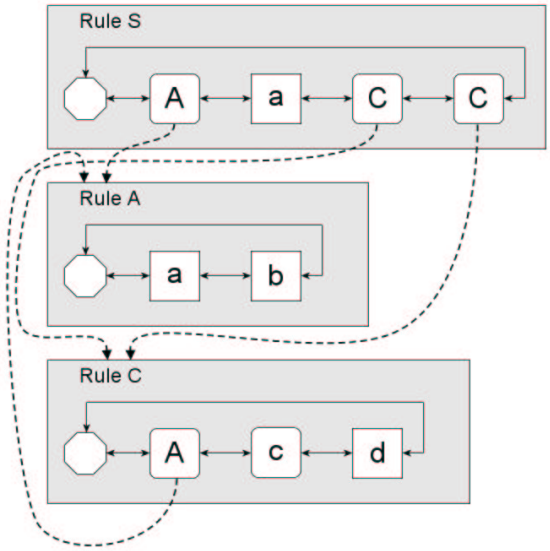


Figure 4.1: Node and rule structure of Sequitur for the grammer in Table 2.1.

Once the digram is found in the hash table, only one step is required to lookup the rule that contains it. This allows for fast replacement of digrams should they be frequently used.

4.2 Instrumenting a Load Instruction

The compiler used to implement this instrumentation technique is the Open Research Compiler (ORC) version 2.0 [ORC]. ORC was developed jointly by Intel Corporation and The Chinese Academy of Sciences. It has been designed to be an open source Itanium Processor Family (IA-64) compiler infrastructure for use in the compiler and architecture research community. ORC is based on the open source Pro64 compiler. It has front-ends for C/C++ and Fortran 90. The back-end includes inter-procedural analysis and optimisations, loop-nest optimisations, scalar global optimisations, and code generation. Linux is the target platform used by ORC.

All implementation for this thesis is done inside the code generation section of ORC. To be able to benchmark the improvement in performance gained by profiling in parallel, two passes of the code have implemented – Traditional and Parallel. The traditional pass inserts the instrumentation code in the same way as any modern profiler. The code is added for each load and is contained within a single basic block. The Parallel pass spans the insertion of the instrumentation code across basic blocks if certain conditions are met. Both passes are detailed below. Both passes are executed

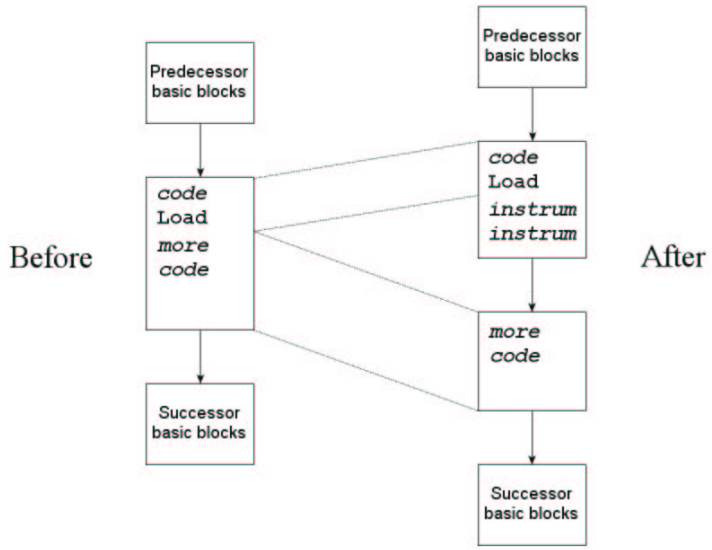


Figure 4.2: Traditional instrumentation technique.

fairly close to the beginning of the code generator. This allows the subsequent optimisation passes present in the code generator to also optimise the instrumentation code. A pass is executed for each region of code in the program. Recall that a region of code is typically a function in a program, which means that these passes only instrument a function (or region) at a time.

4.2.1 Traditional Profiling

The traditional pass adds the instrumentation code presented in Figure 3.4 after each integer load it encounters. This method of adding instrumentation does not span the instrumentation over multiple basic blocks. The code is simply added and treated as any other instruction in the original program.

Due to the fact that the final instruction in the instrumentation is a call branch, all the instructions after the load instruction must be placed in another basic block. Recall that every branch instruction must be the final instruction in a basic block. The original CFG is altered so that the new basic block is the successor of the basic block the instruction originally belonged to. Figure 4.2 explains the process much better than words can.

These new basic blocks increase the size of the CFG by the number of load instructions. This may increase the time needed by any subsequent optimisation passes of the code that require analysis of the CFG, but they do not make a difference to

the final code produced by the code emitter. These new basic blocks are only linked to the original basic blocks once (i.e. single predecessor, single successor), which does not directly alter the final ordering of instructions. However, it may alter the actions taken by subsequent optimisation passes.

4.2.2 Parallel Profiling

The parallel pass adds the same code as the traditional pass does, but in a different fashion. As stated in Section 3.4, the instrumentation code may be spanned over several basic blocks. The decision on whether to span instrumentation over a basic block or not must take many things into consideration. Knowledge of the number of instructions, loops structures and CFG structures are required before instrumentation takes place. To gain this knowledge, two passes of the region are required.

First Pass

The first pass calculates the nest level of each basic block, as well as counting the number of integer load instructions and the total number of instructions. Each basic block has a number of flags associated with it. The two most important flags are "barrier_top" and "barrier_bottom". If "barrier_top" is true, instrumentation code from above cannot be spanned into this basic block. Conversely, if "barrier_bottom" is true, instrumentation code cannot be spanned below this basic block. For example, the head of any loop always has "barrier_top" set to true. In fact, any time a basic block has a different nest level to any of its predecessors, "barrier_top" is true. The reason for this is because any instrumentation must be completely contained within the nest level of its corresponding load instruction. This flag ensures that code does not span loop boundaries.

The final instruction in a basic block determines whether "barrier_bottom" is set or not. If the final instruction is a call, return or loop branch of any kind, "barrier_bottom" is set. To determine whether a conditional branch is a barrier to instrumentation or not, the structure of the CFG is needed. Here is where the 4 cases from Figure 3.2 come into action.

Case A: Attempt to span instrumentation over BB:1 and BB:3. BB:1 has "barrier_bottom" set if BB:2 has one or more integer loads. If this were not the case, the instrumentation would be nested within itself whenever BB:2 is executed. If "barrier_bottom" is not set, calculate which of the two successors should have Stage 2 inserted.

Case B: Attempt to span instrumentation over BB:1, BB:2 and BB:3. The instru-

mentation in BB:2 must be duplicated in BB:3. If either BB:2 or BB:3 have different nest levels than BB:1, BB:1 has "barrier_bottom" set to true.

Case C: Attempt to span instrumentation over all basic blocks. If any BB:2i have different nest levels, BB:1 has "barrier_bottom" set to true. All BB:2i have "barrier_bottom" set because any instrumentation started in BB:2i must be completely contained within BB:2i.

Case D: The conditional branch in BB:1 always causes "barrier_bottom" to be true because its successors have different nest levels.

This pass can be summarised in the form of an algorithm which is presented in Figure 4.3.

Although "barrier_top" and "barrier_bottom" have opposite names, they are not strict opposites. Recall from Section 3.4 that if instrumentation cannot be contained within one basic block, it is split into two stages. If "barrier_top" is set, it cannot contain any Stage 2 instrumentation that its predecessor wants to insert. If "barrier_bottom" is set, both Stage 1 and Stage 2 must be completely contained with this basic block. In other words, "barrier_top" effects its predecessor while "barrier_bottom" effects the current basic block. This concept is also explained in Figure 4.4.

Second Pass

The second pass is where the instrumentation code is added. When this pass comes to a basic block, it first checks to see if it contains a load instruction to instrument. If not, it picks another basic block until all basic blocks have been visited. Otherwise, if this basic block does need to be instrumented, the decision on whether to span this instrumentation over its successors must be made. The first check is the number of instructions in this basic block. If the entire instrumentation can be added to this basic block while not breaking the Instruction Ratio, then there is no need to span. If the Instruction Ratio will be broken, then more checking is required.

Based on the flags "barrier_top" and "barrier_bottom", it is possible to quickly determine whether it is possible to place instrumentation code in its successors or not. If both flags are false, the following actions occur:

Case A: Insert Stage 2 into the correct successor (determined by the first pass).

Case B: Duplicate Stage 2 in both successors.

Case C: Duplicate Stage 2 in all successors.

```
// First Pass
foreach basic block, BB, in region
do
    Calculate BB's nesting level (BB->nl);

    if (BB has a different nesting level to any of its predecessors)
        BB->barrier_top = TRUE;

    if (BB has a different nesting level to any of its successors)
        BB->barrier_bottom = TRUE;

    if (BB has more than 1 successor && BB->barrier_bottom == FALSE) {
        if (any successor has a nesting level > BB->nl)
            BB->barrier_bottom = TRUE;

        determine which Case BB belongs to;

        // Only Case A is handled here. The other cases are handled generically
        // by the above code
        if (BB belongs to Case A) {
            if (any basic block in the conditional branch has a load
                or has a nesting level > BB->nl)
                BB->barrier_bottom = TRUE;
                Calculate which branch is the conditional branch;
        }
    }

    foreach instruction in BB
    do
        BB->nInstructions++;
        if (instruction is an integer load)
            BB->nLoads++;
    done
done
```

Figure 4.3: Pseudo-code for the first pass.

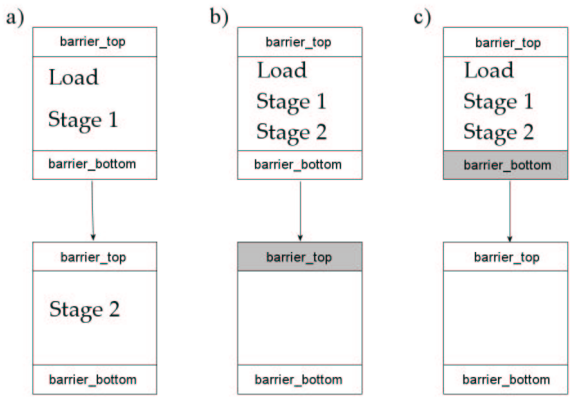


Figure 4.4: How different types of barriers cause instrumentation to be restricted to certain basic blocks.

Case D: Both Stage 1 and Stage 2 must be contained in this basic block, no spanning can take place.

The final instruction in Stage 2 of the instrumentation is the call to Sequitur. If the final instruction in any basic block where Stage 2 is inserted is a branch instruction of any kind, a similar process to that in the traditional pass takes place. The final instruction must be moved to a new basic block so that the call to Sequitur can take place before the branch. If the final instruction in the basic block is not a branch instruction, no changes need to take place. Once every basic block in the region has been visited, the second pass is complete.

This pass can be summarised in the form of an algorithm which is presented in Figure 4.5.

4.2.3 Instruction Scheduler

The original plan for this thesis was to extend the instruction scheduler so that is was aware of the two separate code streams. After implementing the traditional pass and inspecting the code produced, it was noted that the instruction scheduler was already doing a good job of executing the two streams in parallel.

The instruction scheduler in ORC used dependencies to determine which instructions to schedule next. Any instruction that has all of its dependencies fulfilled is a potential candidate to be scheduled. Since the two code streams only have dependencies on themselves, as one dependency from the original stream is fulfilled, so is another dependency from the instrumentation code. The result is the two streams being executed in parallel close to perfectly. On occasions, the types of instruction

```
// Second pass
foreach basic block, BB, in region
do
    if (BB->nLoads == 0)
        continue;

    if (BB violates Instruction Ratio && BB->barrier_bottom == FALSE) {
        switch (BB->case) {
            Case A:
                if (barrier_top is FALSE for both successors) {
                    add Stage 1 to BB;
                    add Stage 2 to non-conditional branch;
                }
                break;
            Case B or Case C:
               if (barrier_top is FALSE for all successors) {
                    add Stage 1 to BB;
                    add Stage 2 to each successor
               }
               break;
        }
    }
    else
        add both Stage 1 and Stage 2 to BB;
done
```

Figure 4.5: Pseudo-code for the second pass.

in each stream cause more dependencies to be fulfilled at a certain point than the other stream. This results in a few cycles containing only instruction from one stream. Although this means that the two streams and not executed in parallel all the time, it is close enough to not cause any problems.

Since the instruction scheduler uses dependencies to choose the next instruction, and there are no dependencies between the two streams, the result is that part of the instrumentation code is moved above the load. This does not cause a problem except for one case. Line 1 of the instrumentation code in Figure 3.4 is where the IP is read. For the profile data to be accurate, this instruction must be in the same instruction group as the load instruction. To make sure this happens, a fake dependency between the load and the IP read is created. This dependency has zero latency, meaning that the two instructions can be executed in the same instruction group without violation. When the scheduler comes across an instruction, it automatically takes any instruction with zero latency to be bundled it with. This small change in the scheduler is all that is needed to ensure that the two instructions are executed simultaneously.

Chapter 5

Experimental Results

To evaluate the performance of profiling in parallel, SPECint2000 (a suite of benchmarking programs) is used. SPECint2000 compiles these programs and then records the execution time to perform a specific task. All benchmarks were executed three times, and the average of those runs calculated. The machine used to run SPECint2000 has a single Intel Itanium processor with a 32kb L1 cache and a 2Mb L2 cache. The clock-speed of the CPU is 666MHz. The machine is fitted with 1Gb of RAM, and a Quantum 8Gb SCSI hard drive. All programs are compiled with `-O3`, which for ORC, enables most optimisations that do not require feedback. The maximum Instruction Ratio for any basic block containing an instrumented load was kept constant at $3:4$ across all programs.

5.1 Preliminary

To display the change in overhead between the traditional implementation and the parallel implementation, three runs of SPECint2000 are required. The first run is the base run where no extra code is added to the programs. For the second run, the tradition method of instrumentation is added to each program. Finally, the third run is for the parallel method of instrumentation.

The problem with doing this is that we are actually measuring two things: the overhead from the profiling method and the overhead from the Sequitur algorithm. The only thing that should be benchmarked is the overhead from the different profiling techniques. The overhead from the Sequitur algorithm distorts the results and makes the amount of change in overhead from the profiling difficult to see. To remedy this problem, the instrumentation code from Figure 3.4 is altered. Line 14, the call to Sequitur, is completely removed. To stop the array from being overfilled, line 7 (where `current_elem` is incremented) is changed to:

```
Line 7          adds    r1    = 0, ce_val
```

This results in zero being added to `current_elem` each load. In other words, the first element of the array is continuously being overwritten. Although in practice this would never happen, for the purpose of benchmarking only profiling, this is the solution.

ORC is still under development, and will more than likely always be under development since it is a research tool. As such, it has a few problems compiling all of the SPECint2000 programs. The five programs that it always compiled and executed correctly are `164.gzip`, `175.vpr`, `181.mcf`, `252.eon` and `256.bzip2`.

5.2 Revised

With the revisions made, SPECint2000 is able to produce a much clearer view of how the different profiling techniques change the amount of overhead. Table 5.1 shows the execution time of each program for the three runs. The difference in overhead between the traditional and the parallel approach to profiling is shown in the "Percentage Reduction in Overhead" column.

Figure 5.1 displays the amount of overhead added to each program for the two instrumentation techniques. The amount of overhead for each program is normalised to the base run.

To better understand the reasons behind the changes in overhead, statistics about each program must be known. Table 5.2 presents these statistics. For each benchmark program, the number of instructions and number of integer load instructions are displayed. The number of opportunities where the instrumentation code may have been spanned over basic blocks is shown in the "Attempts" column. The "Parallelised" column displays the number of times the attempt to span instrumentation was successful. The difference between these two columns is caused by the restrictions placed on where instrumentation can be spanned. Recall from Subsection 4.2.2 that the structure of certain CFGs around the basic block containing the load instruction may force the

Benchmark	**Execution time (seconds)**			**Percentage Reduction**
Program	**Base**	**Traditional**	**Parallel**	**in Overhead**
164.gzip	702	872	852	2.8%
175.vpr	790	1210	1052	20.0%
181.mcf	1232	1812	1591	17.9%
252.eon	730	1104	986	16.2%
256.bzip2	845	1044	948	11.4%

Table 5.1: SPECint2000 benchmark results.

instrumentation to be contained within only one basic block. The top and bottom barrier flags, and the nesting level of each basic block, are what cause this difference.

Another set of statistics that would be advantageous to know is how many times each type of instrumentation is executed. Based on the execution paths taken by the programs, how many times was the spanned instrumentation executed compared with the non-spanned instrumentation. To produce these statistics, an additional level of profiling would be required. Unfortunately, the addition of another level of profiling would change the structure of many regions of code just as these profiling techniques have. The second profiling method would also need to be able to tell the difference between instrumentation instructions and the original instructions in order to know where to add its own instrumentation. So while it would be helpful to have these extra statistics, it is not practical. Instead, inferences must be drawn by combining all the results and statistics together. A complete discussion of what these results represent is given in the next Chapter.

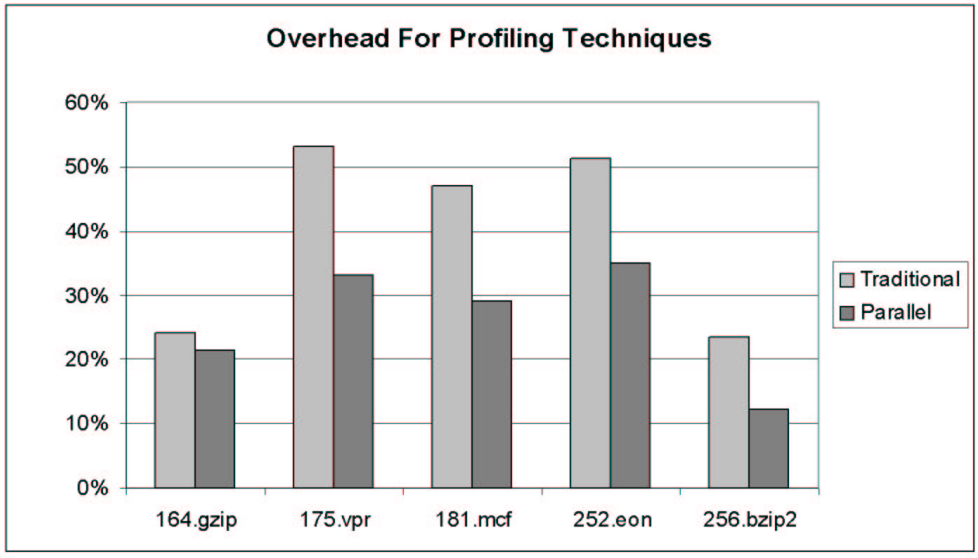


Figure 5.1: Percentage overhead for the SPECint2000 benchmark programs.

Program	Instructions	Loads	Attempts	Parallelised
164.gzip	11981	497	48	10
175.vpr	41786	2463	166	28
181.mcf	3393	272	23	0
252.eon	190485	5385	31	18
256.bzip2	8632	357	24	8

Table 5.2: Statistics for the SPECint2000 benchmark results.

Chapter 6

Discussion

By observing the experimental results from Table 5.1, it is clear that profiling in parallel has an advantage over the traditional profiling approach. In all five benchmark programs, the parallel method of instrumenting code reduced the overhead needed to build a profile of a program. The range of overhead reduction is 2.8% to 20.0% with an average reduction of 13.7

The number of load instructions compared with the number of times the instrumentation was spanned over several basic blocks ("Parallelised" column) is quite low across all benchmark programs. Even the number of candidates that may have been parallelised ("Attempts" column) is only a fraction of the number of loads. The reason for this is the constraints placed on the instrumentation technique. There are far more cases where instrumentation is forced to be confined to a single basic block than there are cases where spanning is permitted. The profiling in parallel technique is rather conservative with respect to where it is applied. Chapter 7 describes ways of improving these numbers.

The program with the greatest reduction in overhead, `175.vpr`, is also the program with the highest number of spanned instrumentations. This is also quite promising. Notice that the number of times instrumentation was parallelised for `181.mcf` is zero and yet there is still a 17.9

The first step in building the stages is to remove the instructions that always produce the same results. These four instructions are moved to the entrance to the region and only executed once. Compare this with the traditional approach, which executes these instructions for every load. For `181.mcf`, this means there are $4 \cdot (272 - 1) = 1084$ less instructions to execute. If some of these instructions exist within loops, the number of saved instructions is multiplied. This can be said about all the benchmark programs – if the instrumentation instructions exist within loops, removing a number of instructions has a much larger impact on performance than

removing instructions that exist outside of loops.

Instrumentation that exists on frequently executed paths of execution (hot data streams) has more of an impact on overhead than instrumentation on infrequently executed paths. Bases on the statistics from Table 5.2, the parallel instrumentation for `164.gzip` was applied more than it was for `256.bzip2`. Cross referencing that with the results in Table 5.1, the inference can be drawn that it is less likely than the sections of code where profiling in parallel was performed did not exist on hot paths. It is more likely that since `256.bzip2` had a greater reduction in overhead even though only eight applications of the instrumentation were spanned, these applications existed on hot paths.

Chapter 7

Future Work

This thesis merely scratches the surface of executing two streams of code in parallel. There are many different ways the ideas and techniques developed in this thesis can be extended and improved. The most obvious next step is to extend the work to implement the entire bursty tracing framework [HC01]. This could then be extended further to the hot data stream prefetching detailed in Section 2.4. The final product would then allow complete flexibility and control by manipulating a few variables and counters.

Even without continuing the work to such an extent, it is possible to develop an analysis method to reduce the number of function calls to Sequitur. This idea was touched on in Subsection 3.4.2. One possible solution is to have a call to Sequitur at the entry point to each region as well as each exit point. This will remove any problems caused by other call branches. Loops are also an issue though, so perhaps it is necessary to also have a function call to Sequitur inside each loop. Depending on the number of loads in each iteration of the loop, this may lead to overfilling the profile array. To fix that, an overflow array or an array that is bigger than the instrumentation thinks it is, could be used during the time that Sequitur is not called. When Sequitur is finally called, even though the array has more data in it than Sequitur allows, it has not been overfilled. So for example, `profile_array_size` could be 1000, but the actual size of the `profile_array` is 1100. This way, as long as Sequitur is called at least every 100 load instructions, the array would never be overfilled. Of course to guarantee this, analysis would need to be performed on each loop to determine exactly how many iterations are performed ahead of execution.

The instrumentation used in this thesis is not re-entrant. In other words, it cannot be nested within itself. This may be necessary if instrumentation is to be spanned over a call branch. Designing such code is one more possible avenue of continuing this work and could very well reduce the overhead substantially.

Another way to extend the work in this thesis is to develop formal algorithms to find the best possible way to span a specific code stream over another code stream. Some techniques for handling branches is described in Section 3.4, but no formal algorithm has yet been developed. It should also be possible to manipulate the instrumentation to better suit multiple load instructions within a basic block. Manipulating the instrumentation that exists within loops should also improve the reduction in overhead. Creating a few implementations of the same instrumentation that can be applied in certain situations to produce optimal results will definitely help.

The concept of executing two independent streams of code simultaneously does not have to be limited to profiling. This could develop into a new form of loop optimisation in which two independent loops are combined and performed completely in parallel.

Further development of ORC to make the instruction scheduler aware of the two streams may improve its ability to explicitly execute the streams in parallel. At present, it does this by default due to their being no dependencies between the streams. If the streams were dependent on each other (even only slightly as in the case of this instrumentation technique where the load instruction must be execute in parallel with the read of the instruction pointer), it would be nice to have the facility to explicitly tell the scheduler that two or even three instructions must be in the same instruction group. This facility does not only have to exist in the code generator, but could be extended to intermediate representation used by the compiler and even to the high level languages such as C and C++.

As is quite evident, the possibilities for extending the simple ideas presented in this thesis are virtually endless.

Chapter 8

Conclusion

This thesis has presented a possible new research area – executing two streams of code simultaneously on a single CPU. The two streams of code referred to in this thesis are instrumentation code and the original program code. The reason behind attempting to parallelise the execution of these two streams is to reduce the amount of overhead usually associated with profiling. Feedback-driven online optimisations require the profiling have a low level of overhead so that overcoming this overhead does not waste the speed-up from the optimisation.

This instrumentation technique of profiling in parallel with the original program has been designed specifically for the Intel Itanium family of processors, and is directed solely at the L1 data cache. By utilising the capabilities of the Itanium processor to execute three instructions simultaneously, it is possible to reduce the amount of overhead required to build a profile of a program by an average of [[**XX%**]].

To increase the changes of instrumentation being executed in parallel, the instrumentation code is split into stages. These stages are then spanned over several basic blocks. The idea is were the original program is unable to fully utilise an instruction group, the instrumentation code can be performed. For some structures of basic blocks, it is not possible for the instrumentation code to be spanned out. This forces the use of the traditional approach of instrumentation that confines the instrumentation to a single basic block.

The instrumentation technique detailed in Chapter 3 is a quite conservative approach to execution in parallel. The technique is fairly limited and there are many cases in which it is unable to span instrumentation over basic blocks. It may be possible by pursuing the ideas in Chapter 7 to reduce this overhead even more.

This thesis has not only developed a way of executing instrumentation code in parallel with the original program code, but has developed a technique of executing two virtually unrelated streams of code in parallel. There is no reason why the techniques

presented cannot be extended to any two streams of code. It is not limited to profiling alone. Not only does this thesis provide a method for reducing overhead by profiling in parallel, it also forms the groundwork for future research in the area of parallel execution on a single CPU.

Bibliography

[AR01] Matthew Arnold and Barbara Ryder. A framework for reducing the cost of instrumented code. *Programming Languages Design and Implementation (PLDI)*, 2001.

[ASV01] A. Edwards A. Srivastava and H. Vo. Vulcan: Binary transformation in a distributed environment. *Microsoft Reseach Tech Report, MSR-TR-2001-50*, 2001.

[CH02] Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. *Programming Languages Design and Implementation (PLDI)*, pages 199–209, June 2002.

[Chi99] Trishul M. Chilimbi. Cache-conscious data structures – design and implementation. Master's thesis, Computer Sciences Dept., University of Wisconsin-Madison, July 1999.

[Chi01] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, June 2001.

[HC01] Martin Hirzel and Trishul M. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. *4th Workshop on Feedback-Directed and Dynamic Optimisation (FDDO)*, pages 117–126, December 2001.

[int] Intel itanium processor family. URL `http://developer.intel.com/design/itanium/family/`.

[Int99] Intel Corporation. *IA-64 Application Developer's Architecture Guide*, May 1999.

[Int02] Intel Corporation. *Intel Itanium 2 Processor Reference Manual*, June 2002.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[NMW97a] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structures in sequences: A linear-time algorithm. *Journal of Artifical Intelligence Research 7*, September 1997.

[NMW97b] C. G. Nevill-Manning and I. H. Witten. Linear-time, incremental hierarchy inference for compression. *Proceedings of the Data Compression Conference (DCC)*, 1997.

[ORC] Open research compiler. URL `http://ipf-orc.sourceforge.net/`.

[SRC02] R. Bodik S. Rubin and T. Chilimbi. An efficent profile-analysis framework for data-layout optimizations. *Principles of Programming Languages (POPL)*, January 2002.

Acknowledgements

First and foremost, I would like to thank my thesis supervisor, Jingling Xue. Without his guidance and immense support, this would not have been possible. I would also like to thank my family, Aldona, Alan, Dana and Anne Sankauskas. Their love and support has helped me greatly. There are many other people who have encouraged and inspired me along the way. They know who they are and I thank you all.