Run-time Adaptation for Multi-Core Execution

Abstract

Contention over shared resources is one of many variables affecting parallel performance on modern multi-core processors. The issue is of particular interest because it is the primary factor limiting the performance of fully independent threads/processes which would otherwise showcase linear speedup with each additional core. Furthermore, addressing the problem is a non-trivial task. Shared-resource contention, while fundamentally a product of architectural design decisions, is significantly influenced by the running workload as well as code-generation decisions made at compile time. This paper investigates the effects of workload selection and code-generation on parallel performance and showcases that there is no single factor or characteristic that can be leveraged to provide a best static compilation or workload selection with respect to parallel performance. Addressing resource contention must be handled in an adaptive manner at runtime. To this end, this paper proposes a hybrid runtime system that uses hardware performance counters to influence runtime code selection. The system is evaluated using a subset of the SPEC CPU2006 benchmark suite and is demonstrated across variety of x86 multi-core architectures.

1 Introduction

While research focused on helping programmers parallelize code is essential to the success of multi-threaded architectures, there are many other issues that also arise in a multi-context system. One particular issue is that of contention over shared resources between multiple simultaneously executing threads. By virtue of many engineering decisions (cost, performance, power), all existing and proposed multi-threaded designs include some number of hardware subsystems that are shared between the various contexts of execution. For example, the Intel Core2 architecture utilizes a shared L2 cache between processor cores, while processors sporting Smultaneous Multi-Threading (SMT) share significant portions of the microarchitecture between execution contexts.

Contention over shared resources is important because it is one of many variables affecting parallel performance on modern multi-threaded processors. The issue is of particular interest because it is the primary factor limiting the performance of fully independent threads/processes. Even if someone develops a method to perfectly parallelize code, the issue of shared resource contention will always limit the possible speedup. Furthermore, addressing the problem is a non-trivial task. Shared-resource contention, while fundamentally a product of hardware design decisions, is significantly influenced by the running workload as well as code-generation decisions made at compile time.

This paper explores shared resource contention in relation to code versioning and dynamic code selection. In particular, this paper shows how compiling the same code using various compiler flags can result in a set of binaries that exhibit different favorable characteristics. One code version may be the clear performance winner when running alone on a system, while a different version may be significantly better when running alongside another application. In fact, this paper shows that the best version depends on the behavior of the paired workload and showcases a system designed to select the best version at runtime by monitoring hardware performance counters.

Specifically, this paper makes the following contributions:

1. An analysis that showcases that different binaries generated with different compilers and optimization flags yields characteristics that are each favorable under different circumstances.

2. A method for statically selecting a set of code versions that can merged into a new binary that provides the best average performance across a given workload.

3. An implementation of a dynamic, runtime system that monitors hardware performance counters and adaptively selects different code versions online depending on the system behavior, leading to better performance than the static solution.

2 Motivation and Background

A substantial amount of previous work on shared resource contention dates back to early work on simultane-
Figure 1. Compiler and Optimization Flags

Table: Compiler and Optimization Flags

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Optimization Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc 3.4</td>
<td>-O0, -O1, -O2, -O3, -Os</td>
</tr>
<tr>
<td>gcc 4.1</td>
<td>-O0, -O1, -O2, -O3, -Os, -march=prescott</td>
</tr>
<tr>
<td>icc 10.0</td>
<td>-O0, -O1, -O2, -O3, -Os, -xT</td>
</tr>
</tbody>
</table>

Figure 1. Compiler and Optimization Flags

ous multithreading (SMT) architectures. At the hardware level, Tullsen et al [5] explored instruction fetch policies designed to enable forward progress of both threads as well limit inter-thread interference. On the topic of code generation, Lo evaluated [2] the effects of various compiler optimizations on SMT performance and provided advice on how to best tune optimizations for code aimed at running on such architectures. There have also been a variety of papers published on OS scheduling for SMT architectures [1, 4] as well modeling SMT resource contention [3]. Similar topics have been explored with respect to other multi-threaded architectures, with the general principles being the same. [Discuss CMPs more; cite references / modify last sentence]

[Add additional paragraph here that bridges into next section]

2.1 Code Generation Methodology

Rather than looking at specific compiler optimizations, this paper explores code versions. A code version is simply a region of code compiled using a specific compiler with particular set of optimization flags. Furthermore, this work uses functions as region boundaries, although one could in practice define a region to be something else such as a basic block, hyperblock, or dynamic trace. In total there are 27 versions considered, resulting from using three compilers and nine different sets of optimization flags. Figure 1 shows the different optimization flags used with each compiler. The optimization flags in the third column correspond to those in the second column plus additional flags that allow the compiler to generate code specifically for the newest Intel processors. Specifically, these flags enable the use of MMX and SSE instructions as well as code that is aware of modern microarchitecture features such as the hardware prefetcher.

Given a set of versions, this paper is motivated by the following issues:

1. The compilation that provides the best single-threaded

performance may not perform the best when part of a shared workload.

2. Moreover, there may not be a best compilation for the shared case. The best compilation largely depends on the characteristics of the other running threads.

2.2 Case Study: 400.perlbench

To demonstrate the issues presented above, consider the data shown in Figure 2 for the SPEC CPU2006 benchmark 400.perlbench. Each graph shows the performance characteristics for a specific function with respect to execution time (x-axis) and number of L2 cache accesses (y-axis). In many cases, the fastest version also has the fewest number of L2 cache accesses. However, this is not universally true across all of the shown functions. Given that the L2 cache is the primary element shared between cores on a dual-core Intel Core2 Duo processor, the number of cache accesses
3 Dynamic System

Since contention over shared resources is dependent upon the running workload and the runtime characteristics of the paired threads, a dynamic solution to the problem is the most appropriate. This section details such a system that was implemented as part of this work.

3.1 Overview

The dynamic system implemented in this work consists of two primary components. The first component consists of an offline linker that is capable of creating a binary containing multiple compiled versions of a set of program functions. The second component is a runtime system that monitors the performance of the system using hardware performance counters and then dynamically selects between the different function versions.

The offline linker is a straightforward modification of GNU ld. The modified linker takes in a configuration file which lists the desired function versions as well as a pointer to a directory tree containing compiled object files generated with the different compilers and optimization flags. The linker configuration file also lists which function versions should be installed by default. The linker then generates a new binary by combining the desired functions from the various object files and renames the functions so that each duplicate function has a distinct name. Under the original function name, the linker generates a stub function that consists solely of a `jmp` instruction with a hardcoded target address to the default function version.

3.1.1 Runtime Component

The aim of the runtime component is to monitor the hardware performance counters of the system and dynamically modify which function versions are selected by altering the jump target address of the stub functions created by the offline linker. The prototype implemented for this paper was built on Linux using OProfile[REF] to gather the performance counter data and Intel’s Pin[REF] as the basis of the runtime selection framework.

For this work, the standard OProfile daemon was modified such that the system no longer generates log files with raw performance counter results, but instead maintains a running in-memory scoreboard listing which system processes are generating the most hardware events. This scoreboard also includes the list of instruction addresses generating these events as well as a running total of events generated. Additionally, the scoreboard is designed to show temporal behavior, and therefore the results are reset every 500 ms. A simple API is also provided allowing other system processes to access this scoreboard data.
The actual runtime system logic is built as a Pin tool which is dynamically inserted into an application at runtime. While this approach was used for this prototype, a dynamic instrumentation system such as Pin is not a requirement for this technique in general. In particular, it is also possible for the offline linker discussed above to directly include the runtime system code in the generated binary itself.

The runtime code installs a timer interrupt and sets up an appropriate signal handler. As the timer fires, the handler uses examines the OProfile scoreboard and determines if a selection decision should be made. The specifics of how this selection decision is made is detailed in next section.

3.2 Dynamic Selection Methodology

For this paper, two dynamic selection approaches were considered. In both cases, the offline linker is used to generate a combined binary which includes three versions for each of the top 20 functions within the benchmark. The three versions differ primarily in their cache usage as determined through initial profiling. At runtime, the selection logic switches between these three versions as a response to system L2 cache events.

3.2.1 Greedy Selection

The greedy selection approach is provided with profile data listing the average L2 cache misses expected for each of the three versions when running alone. At runtime, if the system notices that it’s individual L2 cache misses are greater than normal the system will switch to a lower cache usage code version in order to optimize performance. This selection approach only considers its own individual L2 miss performance and ignores the effects it is having on the system as a whole. This approach will always switch back to the highest performing version regardless of cache usage after a fixed interval of time. It will only throttle down the cache usage to a lower performing version after again sensing significant L2 cache contention.

3.2.2 Cooperative Selection

The cooperative selection approach looks at the scoreboard data in its entirety and attempts to reduce its effect on the system. This selection approach is provided with average L2 cache misses when paired with other applications, and will throttle back its cache usage if the system as a whole is exhibiting miss rates greater than this average of if the individual benchmark itself is generating more than 75% of the system’s total misses. This approach only reverts back to higher performing, higher cache usage versions of code after the system average miss level falls within the average range.

4 Problem Analysis: SPEC CPU2006

[Pre-analysis of the chosen SPEC benchmarks. Show performance and cache metrics for code versions, paired performance data, and discuss dynamic potential using invocation information]

5 Experimental Results

[Show results for running benchmarks with our dynamic system across several architectures – dual-core, quad-core, etc]

6 Analysis

[Analyze experimental results and relate back to SPEC pre-analysis]

References

Figure 4. Invocations